Universidad Carlos III de Madrid

Desarrollo de software de videovigilancia para sistemas embarcados distribuidos con ICE.

PROYECTO FIN DE CARRERA.

INGENIERÍA TÉCNICA DE TELECOMUNICACIÓN: TELEMÁTICA

AUTOR: *Raúl Castro Fernández*

TUTORA: *Dra. Marisol García Valls*

Agradecimientos

En primer lugar quiero agradecer a Marisol, mi tutora durante el proyecto, la dedicación y la ayuda prestada, especialmente por la rapidez récord a la hora de contestar e-mails, y por los consejos que me ha dado para enfocar el proyecto y la realización tanto de esta memoria como del trabajo al completo, así como por lo que he aprendido con ella durante este tiempo.

Después no pueden faltar los agradecimientos a mi gente, a los de siempre, que me han soportado fines de semana sin salir por una cosa u otra durante algunas temporadas, o que me han aguantando hablando de cosas de las que no tenían ni idea porque necesitaba organizarme la cabeza, pero que al fin y al cabo siempre han estado ahí durante toda esta carrera que termino ahora. Me ahorro los nombres porque ellos saben de sobra quiénes son.

Por supuesto, no pueden faltar agradecimientos a Mike y Álvaro, dos piezas claves durante la carrera. A Mike por ser la pareja de prácticas y clases perfecta, además de por los innumerables momentos pasados desde el inicio de todo, que ahora se divisa lejano. A Álvaro por tener esa chispa en la cabeza que nos caracteriza y que hizo que congeniáramos desde el momento que tuvimos contacto, por esa ansia de aprender, por los gustos que nos unen y por lo que me ha motivado todo este tiempo aunque él no lo sepa.

En general a toda la gente que me ha ayudado de una forma u otra a terminar la carrera.

Resumen

En este proyecto se pretende estudiar la tecnología de middleware **Ice** de ZeroC, y más concretamente el servicio **IceStorm** de publicación-suscripción que implementa. El trabajo contiene primeramente una descripción de esta tecnología y su comparación teórica con el estándar **DDS**. En dicho estudio se introducirán las características claves de ambas tecnologías, en sistemas de propósito general, y en sistemas embarcados.

Tras este capítulo se muestra la aplicación de videovigilancia (aplicación de demostración) que se ha desplegado en un entorno distribuido interconectado en una LAN, donde la comunicación se realiza mediante **IceStorm**. Se ha estudiado en esta parte los problemas que planteaba el desarrollo de la aplicación de demostración, sobre todo en los aspectos de interconexión software de distintas tecnologias, y se exponen las soluciones que se ha dado a cada uno de ellos.

Tras justificar todas las decisiones tomadas en el desarrollo de la aplicación y mostrar las partes importantes de ésta, el proyecto presenta un capitulo adicional donde de forma básica se evalua la tecnología **Ice/IceStorm** en un ambiente distribuido.

Finalmente el capítulo quinto describe las posibles mejoras de la aplicación, trabajos futuros, y se dan una serie de conclusiones sobre las tecnologías utilizadas durante la implementación de la aplicación de prueba y durante el estudio del middleware **Ice**.

Como anexos se incluyen de forma detallada la instalación del entorno completo con el cual se ha probado la aplicación y donde se ha evaluado **Ice**.

Índice

Índice de figuras y tablas

Capítulo 1: Introducción y Motivación.

1.1. Introducción

Los sistemas distribuidos tienen una gran importancia en la sociedad actual debido, fundamentalmente, a la posibilidad que ofrecen para comunicar remotamente a usuarios y a éstos con los servicios a los que tradicionalmente acudía de forma presencial. El auge de Internet posibilita aumentar la capacidad de cómputo de los servicios que se ofrecen en la red que son utilizados por multitud de personas y cuya carga se distribuye, en muchas ocasiones, entre muchas máquinas. Ante semejante entorno, donde es evidente que existen multitud de nodos heterogéneos, es necesario utilizar técnicas para saltar los obstáculos que a priori plantea la comunicación entre distintas plataformas hardware y software. El middleware tiene un papel fundamental ante este panorama ya que es la capa de software que permite esconder la heterogeneidad de las plataformas que comunican.

Es muy importante tener en cuenta que las características de los nodos en red pueden ser extremadamente diferentes. Podemos encontrar desde nodos embarcados donde la limitación de los recursos computacionales es el principal problema a salvar, hasta nodos de propósito general (por ejemplo, servidores, ordenadores de sobremesa, etc.) donde se dispone de todos los recursos necesarios para llevar a cabo tareas intensivas de computación.

El caso de los sistemas embarcados, los que se definen como sistemas muy asociados con el entorno con el que interactúan o al que monitorizan, tiene algunos requisitos especiales puesto que aparte de tratarse normalmente de sistemas con recursos limitados, suelen requerir características de tiempo real. El middleware **Ice** se presenta como un middleware bueno para sistemas embarcados por su bajo consumo en recursos, pero no posee características de tiempo real. El caso de los sistemas de tiempo real, que se definen como aquellos de los que se espera cierto determinismo en su respuesta (determinismo temporal, por ejemplo), requiere de soporte, ya sea hardware, o a nivel de sistema operativo que proporcione dichas facilidades. El middleware de tiempo real también ofrece estas facilidades. El estándar **DDS** es un ejemplo.

El presente trabajo forma parte de un trabajo más amplio sobre middleware de tiempo real para entornos embarcados en red. En este marco es

importante trabajar sobre técnicas de tiempo real para ser incluidas dentro del middleware como en el estudio de diferentes tecnologías middleware existentes tanto específicamente para tiempo real (por ejemplo **RT-CORBA** y **DDS**) como para sistemas embarcados (**ICE**).

Tecnologías y paradigmas middleware

Siguiendo el hilo de la introducción, merece la pena mencionar los sistemas que hacen un uso más estricto y más eficiente de middleware en tiempo real actualmente, ya que dichos sistemas nos rodean y son muy variados. Los sistemas electrónicos de los coches por ejemplo, que hacen que salte el airbag a tiempo, o que se active el sistema de bloqueo de frenada (*ABS*), nos da una idea de los requisitos que necesitan dichas aplicaciones. Los instrumentos de cualquier aeronave toman, normalmente, información de distintos dispositivos colocados a lo largo de ésta, conformando sistemas embarcados distribuidos. Dicha información como es de esperar, también es ofrecida por middleware de muy altas prestaciones.

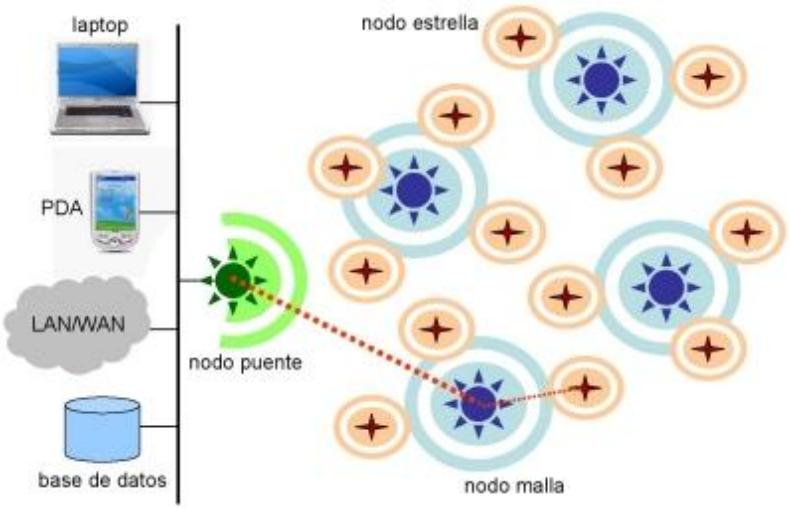


Figura 1: Ejemplo de una infraestructura con sistemas distribuidos embarcados. Red de Nodos.

Dentro del problema de la infraestructura, suele ser una tarea laboriosa la elección de un middleware correcto para una aplicación concreta. Existen muchas alternativas y cada una tiene sus ventajas y sus respectivos inconvenientes, relacionados casi totalmente con el dominio de aplicación y sus requisitos específicos. Las tecnologías de middleware más extendidas hoy por hoy para aplicaciones de propósito general y sin unos requisitos muy severos, son quizá, **Java RMI** (Remote Method Invocation), **Microsoft .NET**, que provee de tecnología middleware que reemplaza el antiguo **DCOM**, y se podría añadir **SOAP**, más orientado a "*web-services*". Por supuesto también todo el middleware de infraestructura (en gestores de bases de datos, etc), el cual ofrece comunicación básica y algunas otras características, está muy extendido y es ampliamente utilizado pero en todo caso no va a tratarse en este proyecto.

En ámbitos más grandes, orientados a aplicaciones empresariales ha sido ampliamente utilizada la tecnología **CORBA** en muchos tipos de aplicaciones, desde aplicaciones de banca, sitios web, hasta redes de sensores quizá algo primitivas. **CORBA** es un middleware orientado a objetos que permite la llamada a métodos remotos de un modo algo similar a que **Java RMI**, que es una tecnología mucho más moderna. **CORBA** ha sido por tanto uno de los middleware referentes durante mucho tiempo.

A todas estas tecnologías se une el hecho de que durante la década de los 80, explotan los sistemas operativos de tiempo real, motivados por la amplia utilización de dispositivos embarcados en coches, aviones, medios de transporte en general. Por supuesto, también es necesario para los sistemas que necesitan requisitos de tiempo real la posibilidad de comunicación por middleware distribuida, por lo que los middleware más usados en ese momento comienzan a sufrir ciertas deficiencias. Este hecho se subsana cuando se decide incluir características de tiempo real en el propio middleware. Nuevamente, uno de los primeros fue la especificación que se hizo de **CORBA** para tiempo real, **RT**-**CORBA**, muy utilizada en un gran número de aplicaciones.

La especificación **RT-CORBA** posee numerosas implementaciones que son utilizadas actualmente en campos que van desde el militar hasta el financiero, concretamente aviones militares, y sistemas de transacción de banca han hecho un uso importante de **RT-CORBA.**

En el campo de los sistemas embarcados, el middleware también cobra

un papel importantísimo, puesto que suele ser la unión entre los distintos dispositivos que conforman una red distribuida. Para optimizar estos escenarios, se han desarrollado desde tecnologías a nivel físico, hasta optimización de los middleware de tiempo real, surgiendo nuevos paradigmas, como el de publicación-suscripción que se ve más adelante.

Es fácil ver que con el tiempo, y la amplitud de aplicaciones existentes, los requisitos de las aplicaciones se hacen más numerosos, y el middleware se ve forzado a renovarse continuamente, para seguir sirviendo de infraestructura de los sistemas distribuidos.

CORBA es una tecnología antigua, que ha ido decayendo por el hecho de no tapar todas las necesidades actuales, o porque para cumplir con esas necesidades la aplicación adquiere una complejidad muy alta, que en muchos casos deja de ser rentable mantener, sobre todo si se tiene en cuenta que se utiliza en ambientes donde la escalabilidad y robustez suelen ser factores importantes. Este hecho ha provocado que el mercado evolucione, y actualmente se está empezando a utilizar middleware que implementa el paradigma de publicación-suscripción. Es la nueva tendencia.

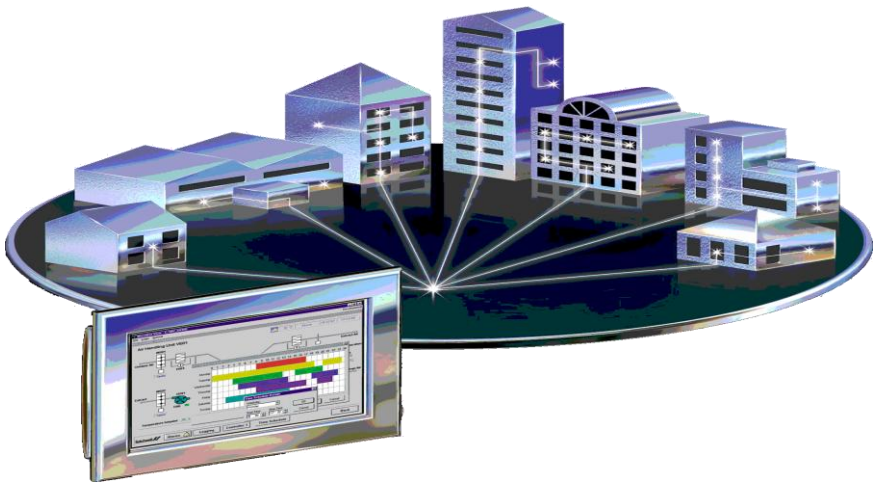


Figura 2: Ilustración de control distribuido de edificios.

Este nuevo paradigma, el de la publicación-suscripción casa perfectamente con aplicaciones como las redes de sensores, donde un número N de nodos necesitan publicar cierta información (“*publishers*”) a cierta o ciertas

estaciones base, que serían en este caso los "*subscribers*". En este caso, en general es más eficiente un paradigma de publicación-suscripción que uno orientado a objetos, porque es directamente más aplicable la idea a la aplicación, además de imponer menos dependencias entre las partes de la misma, o porque el grado de escalabilidad que permite el nuevo paradigma suele ser más alto que el que nos ofrecería un middleware orientado a objetos, entre otros motivos. Como se ha comentado, la escalabilidad es uno de los factores más importantes en sistemas distribuidos, por esto la importancia de este paradigma.

Por supuesto, en el paradigma se deben soportar entornos con requisitos de tiempo real. En otras aplicaciones sin embargo, parece seguir siendo suficiente un middleware de propósito general orientado a objetos, aunque como se comenta, se empieza a ver cómo las empresas y las aplicaciones tienden a utilizar sistemas basados en publicación-suscripción. Se une el hecho de que muchos de los sistemas distribuidos que están floreciendo se sustentan sobre dispositivos embarcados, lo que hace que requisitos como el rendimiento o el uso eficiente de recursos, cobren más importancia.

En esta situación, se encuentran dos tecnologías relativamente nuevas, que implementan ambas un paradigma de publicación-suscripción.

En primer lugar **DDS**, un estándar del OMG, tecnología como se dice relativamente nueva, pero ampliamente testada e implementada por numerosas empresas. La implementación de RTI, quizá la más extendida de todas, y a campos tan críticos como el militar, el aeroespacial o el de la seguridad será la que se tratará en el siguiente capítulo.

Por otro lado, un middleware relativamente reciente y versátil es **ICE**, de la empresa ZeroC. Es un middleware orientado a objetos, basado en **CORBA**, que pretende ofrecer lo mismo que éste último pero simplificando todo el proceso de desarrollo. Una peculiaridad especialmente importante para el desarrollo de este proyecto es el hecho de que **ICE** incorpora varios servicios, los cuales extienden su funcionalidad en varios ámbitos. En especial, **IceStorm** es un servicio que implementa el paradigma de publicación-suscripción en **ICE**, lo cual hace sumamente interesante el estudio de esta tecnología, que en contraste con **DDS**, no incorpora características de tiempo real, y no está tan testada ni extendida como ésta.

1.2. Motivación

Tras la pequeña introducción anterior se pasa a comentar los objetivos y la motivación que persiguen la aplicación y el estudio que se desarrolla en este proyecto.

Como se ha mencionado en las páginas anteriores, **ICE/IceStorm**, y **DDS**, son dos tecnologías middleware que se están empezando a utilizar actualmente y que están siendo, en el caso de **DDS** aceptadas progresivamente y bastante probadas; en el caso de **IceStorm** quizá su alcance sea a dia de hoy bastante menor. La finalidad del proyecto es evaluar en ciertos aspectos la tecnología **Ice/IceStorm**, para lo cual se ha planteado una demo de videovigilancia remota, en la que se utilizará entre otras tecnologías, **IceStorm**, de modo que se pueda estudiar su funcionamiento.

Además, es importante incidir en ciertas características que se encuentran en sistemas distribuidos, como son la heterogeneidad de plataformas que existe. Tanto **DDS** como **ICE** están preparadas para este problema, ofreciendo ambas dos lenguajes que permiten definir datos, métodos, funciones, etc., de forma que sea independiente de plataforma el resultado final. Para emular este escenario, la demo de videovigilancia tiene partes programadas en lenguaje **Java**, y otras partes en **C++**.

Concretamente, la aplicación requiere envío de vídeo a través de una red de área local, entre dos ordenadores que son los que implementan el "*publisher*", (publicador) y el "*subscriber*" (suscriptor) de la comunicación, a través de **IceStorm**. El requisito principal de la aplicación es que el vídeo se visualice en ambos ordenadores, y es deseable que se vea en tiempo real (aunque con ICE más bien se consigue calidad de servicio muy limitada) en ambas maquinas. Retrasos en alguno de los dos ordenadores suponen problemas a los que hay que buscar solución, por ejemplo, muestreando la información que se visualizará, cambiando el canal, o cambiando el middleware e el lenguaje de programacion si finalmente se concluye que merma el funcionamiento de estas aplicaciones.

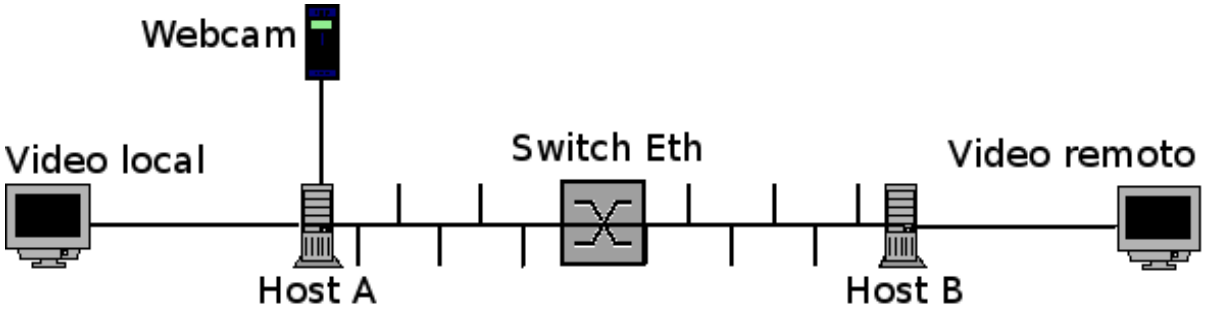


Figura 3: Esquema de la topología de la aplicación demo de vidiovigilancia

Tras realizar la aplicación de videovigilancia, que permite un entendimiento de la tecnología **IceStorm**, como parte adicional (aunque no es en ningún caso la parte central de este trabajo), se realiza una evaluación básica de middleware para poner a prueba la tecnología en función del lenguaje de programación utilizado para distintas condiciones de transmisión de información. Al igual que en el caso de la aplicación, se comentará más adelante, pero básicamente estas pruebas estarán desarrolladas una en el lenguaje de programación **Java** y la otra en el lenguaje **C++**, y cada aplicación estudiará el rendimiento de **IceStorm** con situaciones adversas en cuanto a memoria y con situaciones adversas en cuanto a consumo de procesador.

El fin, por tanto, es observar el funcionamiento de **IceStorm** y observar su comportamiento en ambientes distintos y algo adversos.

Es importante destacar que todas estas aplicaciones se probarán sobre equipos de propósito general con un sistema operativo común. Sería interesante contrastar los resultados con los que se obtendrían si se probaran sobre un sistema de tiempo real, lo cual podría ser interesante como trabajo futuro.

1.3. Estructura del Proyecto

A continuación se pasa a explicar la estructura que seguirá este proyecto y los temas que se irán tratando a lo largo de los capítulos.

Después de este capítulo introductorio, se encuentra el capítulo "*Estado del Arte*", en el que se repasará de forma profunda las características de las tecnologías middleware que más tarde se compararán, RTI **DDS**, y ZeroC **ICE**. Es importante enmarcar primero los ámbitos de aplicación de ambas tecnologías para poder entender las características que las definen. Además, se ofrecerá información relativa a los antecedentes de ambas cuando corresponda. Tras esta información, lo siguiente que se encuentra será la comparación teórica que se realiza entre ambas tecnologías y enfocada a varios planos, como se verá en el capítulo.

El siguiente capítulo trata la fase de diseño, las decisiones tomadas, y la justificación de éstas, con respecto a la aplicación desarrollada. Se verán por separado los problemas que ha supuesto el desarrollo de la aplicación, y la solución que se le ha dado a cada uno de ellos. Se prestará especial atención a la parte relativa a la comunicación, que es la que envuelve a **Ice** y **IceStorm**, y la que tiene por tanto una mayor importancia para este proyecto.

Tras explicar los detalles de la aplicación de videovigilancia, se entra en otro capítulo, que pretende ofrecer un estudio de **IceStorm** con respecto a algunas variables, en el cual se medirá el rendimiento de esta tecnología, enfrentándola a varias condiciones de interferencia. La finalidad de la aplicación de prueba no es otra que estudiar el funcionamiento de **Ice**/**IceStorm** y su integración en un ambiente heterogéneo, y la finalidad que persigue este pequeño estudio es observar el comportamiento del middleware ante situaciones adversas, recabando información que pueda ser útil.

Para finalizar, el último capítulo habla de posibles trabajos futuros y de otras opciones que podrían haberse seguido a la hora de la realización de este proyecto, como sería el despliegue de la aplicación distribuida sobre un sistema operativo de tiempo real, qué ventajas hubiera supuesto, cómo habría variado el

comportamiento, etc. Además, se incluyen las conclusiones a las que se ha llegado a lo largo de todo el proyecto, en los distintos temas que se tocan.

Las últimas páginas de este proyecto se destinan a un anexo en el cual se explican las tareas relativas a la instalación de las herramientas utilizadas, a la configuración de éstas, así como a una descripción de todas aquellas cosas que no se han tenido en cuenta durante el texto, pero que pueden ser relevantes a la hora de desplegar una aplicación como la desarrollada.

Capítulo 2.
Estado del arte.

2.1. Introducción DDS (RTI) y ICE (ZeroC)

En este capítulo introductorio se tratarán principalmente dos tecnologías middleware, **DDS** y **ICE**. **DDS** es un estándar del OMG, y se estudiará la implementación que hace la empresa RTI de éste. Por otro lado, **ICE**, middleware desarrollado por la empresa ZeroC y con licencia GPL. Primero se realizará una introducción a cada una de estas tecnologías, para después centrar la atención en aspectos más concretos de ambas, de modo que se tenga claro lo necesario para realizar una comparación teórica entre las dos.

Ambas tecnologías, tienen en principio unos campos de aplicabilidad bastante alejados, centrándose cada una en un tipo de aplicaciones concretas. No obstante, la motivación de las dos comparaciones que se realizarán es precisamente estudiar las ventajas e inconvenientes que pueden presentar en los distintos tipos de aplicaciones. Por ello se han comparado en dos ámbitos distintos aunque no incompatibles.

Por un lado, se tratará la posibilidad de utilizar estas tecnologías en dispositivos embarcados, así como ver qué dispositivos soportan estas tecnologías, cómo lo hacen, que características aporta cada tecnología, y qué ventajas o inconvenientes presentan cada una.

Por otro lado, dejando de lado la aplicabilidad a sistemas embarcados, se verán los parecidos y diferencias de ambas tecnologías, cuál es su uso habitual, hacia qué campo están enfocadas ambas en su uso más habitual, una vez más, qué características diferencian ambas tecnologías, en qué campos conviene aplicar una u otra, en definitiva, un repaso de ambas tecnologías que dé paso a una conclusión final.

Para realizar ambas comparaciones, se debe primeramente observar su naturaleza, así como la motivación por la que fueron desarrolladas, y los antecedentes que existían previamente. Para ello, se presenta a continuación un resumen de los objetivos que persiguen, repasando algunas de las características y posibilidades que presentan, para más tarde entrar en detalle con algunas características concretas que son las que marcan las diferencias en ambas

vertientes de la comparación, tanto para sistemas embarcados como para un uso más general.

2.2 Introducción a DDS (Data Distribution Service)

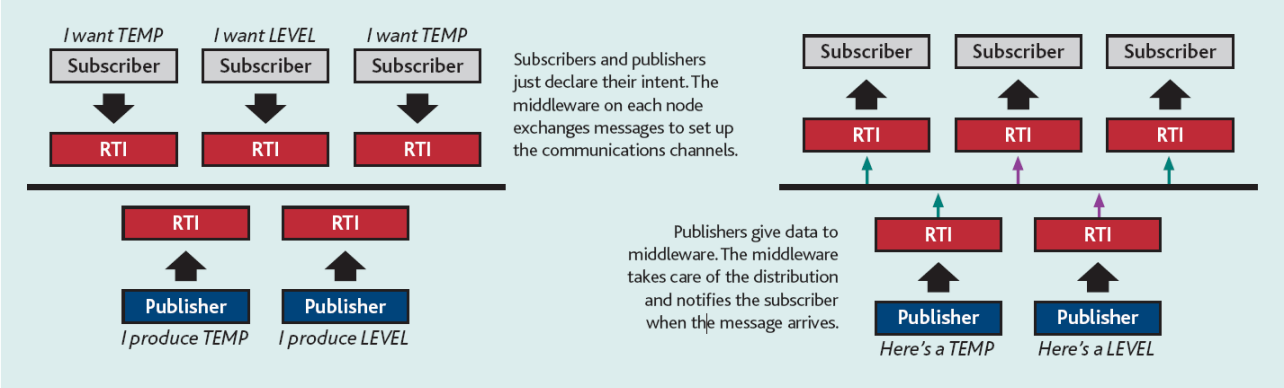
Estándar del OMG (Object Management Group), se define como middleware de tiempo real de publicación-suscripción centrado en los datos. Son varias las empresas que implementan este estándar, particularmente se centrará la atención en la implementación de RTI, aunque esto no supone ningún inconveniente, puesto que todas las implementaciones se basan en el estándar definido por OMG. Las diferencias entre las distintas implementaciones son sutiles y no son objeto de estudio de este documento, basta con tener claro que cada implementación suele añadir alguna característica al estándar, pero que no modifican éste en absoluto.

El objetivo de **DDS** no es otro que facilitar el desarrollo de aplicaciones distribuidas en tiempo real desplegadas en un medio en el cual tanto las plataformas, como los lenguajes en los que se implementa la solución, sean heterogéneos. Esto se consigue porque los datos que intercambia la plataforma **DDS** están definidos en un lenguaje que provee de independencia de plataforma. El lenguaje es **IDL** (Interface Definition Language), y es también un estándar de OMG.

Cabe destacar la importancia de tener un middleware que sea realmente independiente de plataforma y lenguaje, puesto que esto facilita la implantación de la tecnología en distintos dispositivos, la escalabilidad del sistema como se comentará más adelante, y además será clave, junto a otras características de **DDS** para entender por qué es realmente aplicable a sistemas embarcados.

Decimos que **DDS** es un middleware de publicación-suscripción. En el paradigma de publicación-suscripción se definen varias entidades, un "*publisher*", que es el encargado de enviar los datos a la plataforma, y un "*subscriber*" , cuyo cometido es recoger de la plataforma aquellos datos a los cuales esté suscrito. En este contexto, un "*publisher*" publica un "*topic*" al que puede estar suscrito: ninguno, uno o más de un "*subscriber*", es decir, ambos extremos de este paradigma son independientes entre sí. No es necesario para que un "*publisher*" haga su función que haya un "*subscriber*" escuchando en otro extremo. De hecho puede haber N "*subscribers*" que tomen la información del mismo "*publisher*", por eso se dice

también que es una comunicación punto a multipunto, o multipunto a multipunto en el caso de que sean varios "*publishers*" los que publican para varios "*subscribers*".



The publish-subscribe model takes care of channel configuration and data distribution for the application.

Figura 4: Diagrama funcional de la infraestructura RTI DDS. Publish-Subscribe

Esta independencia inherente del paradigma de publicación-suscripción es una de las características clave de **DDS**, se dice que es una tecnología centrada en los datos, precisamente por esta característica que consigue un desacoplamiento muy grande en la aplicación. Además, esta independencia entre ambas entidades supone no sólo una alta flexibilidad a la hora de desarrollar en distintas plataformas o con distintos lenguajes, sino que también potencia la escalabilidad del sistema. Esta escalabilidad puede entenderse pensando que en cualquier momento se puede añadir un "*publisher*" o un "*subscriber*" al sistema sin tener que tomar en cuenta el resto de elementos que haya en dicho sistema. (En **DDS** concretamente sólo se debe prestar atención a que la nueva entidad que es introducida en el sistema se encuentre en el mismo dominio que aquellas identidades con las cuales se pretende la comunicación.)

De esta forma se tiene que **DDS** proporciona una alta versatilidad a la hora de implementarse en un sistema distribuido por dos motivos principales, uno es que la definición de los datos intercambiados se realiza sobre **IDL** que proporciona independencia de lenguaje y plataforma, y la otra es que además, al ser una tecnología centrada en los datos, basada en el paradigma de publicación-suscripción, ambos extremos son independientes entre sí, proveyendo además de una gran facilidad de escalabilidad.

En este punto se tiene un sistema de publicación-suscripción centrado

en los datos, pero no se ha hablado nada de la característica estrella de la tecnología **DDS**, el tiempo real.

La diferencia entre un sistema de propósito general y uno en tiempo real radica en que, aunque ambos tienen que cumplir cierto objetivo, el sistema en tiempo real debe realizarlo de un modo determinista, y en unos márgenes de tiempo previamente prefijados, siendo en ciertas situaciones de gran importancia determinar lo más finamente posible cuando va a comenzar cierta tarea y cuando va a terminar. En cambio, en un sistema de propósito general se busca cumplir con cierta tarea sin atender de manera estricta estos tiempos. Hay muchos matices que hacer sobre esta diferencia, pero con este atisbo es suficiente para comprender cómo las características que ofrece **DDS** facilitan el cumplimiento de tiempos deterministas, es decir la característica de tiempo real.

DDS utiliza mecanismos de calidad de servicio QoS (*Quality of Service*) para proveer a las entidades de ciertas restricciones que desembocan en un determinismo de la aplicación.

Este determinismo se hace patente tanto a la hora de publicar datos, como a la hora de que un suscriptor acceda a ellos, es decir, se tiene una serie de parámetros de calidad de servicio que junto al hecho de que la plataforma sea centrada en los datos, proveen un sistema para indicar cada cuanto deben publicarse los datos, o cada cuanto deben recogerse. También existen QoS que permiten dotar a la plataforma de mecanismos de redundancia, o de fiabilidad, así como otras muchas posibilidades y características que ofertan al sistema y permiten ajustarlo de un modo muy granular.

*La siguiente tabla muestra los parámetros de calidad de servicio (QoS), más importantes de la tecnología **DDS**.*

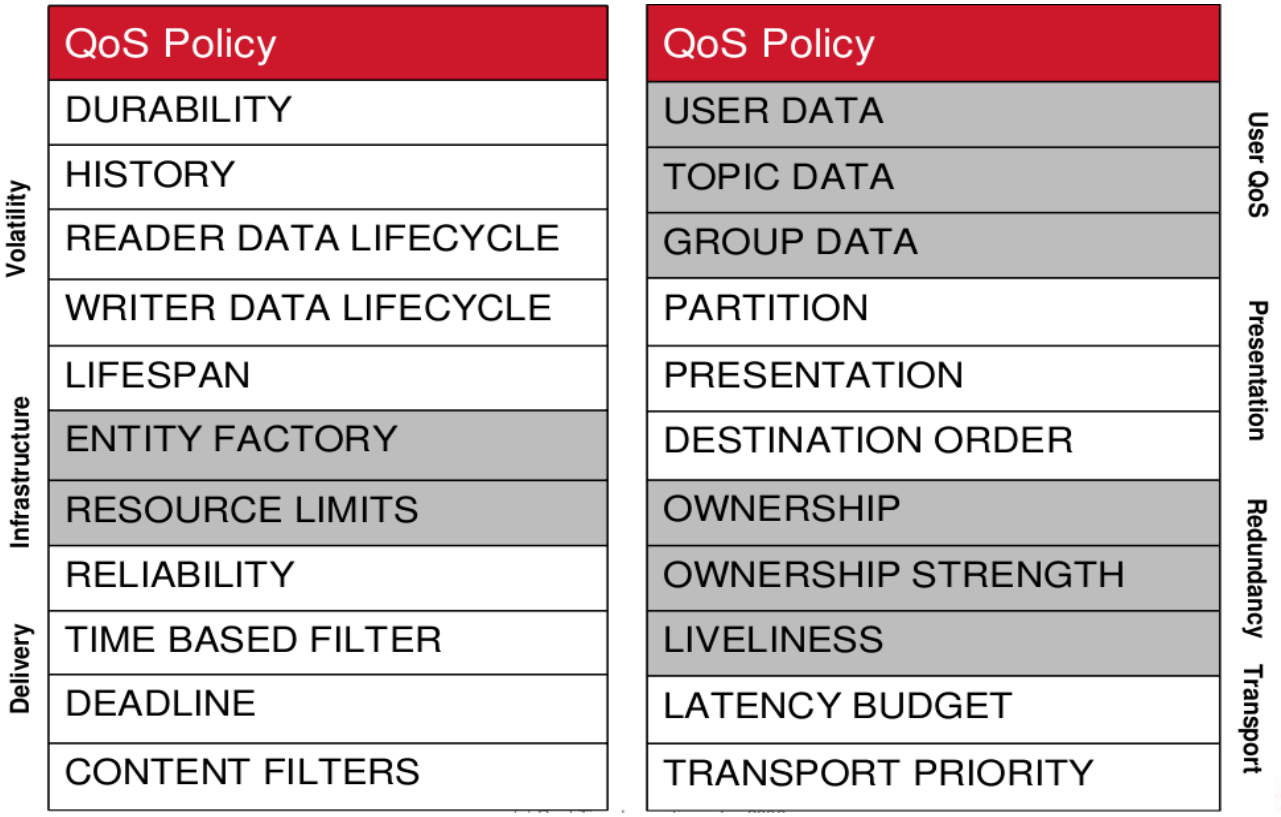


Figura 5: Tabla de los QoS de DDS

Una característica importante de **DDS** y que aún no se ha comentado es el modelo de intercambio asíncrono de datos que ofrece. Un "*publisher*" puede publicar ciertos datos, y es posible que no haya ningún "*subscriber*" en ese momento para recibirlos, pero mediante QoS se puede decidir si esos datos que están "perdidos" en la plataforma pueden ser o no accedidos más tarde por un "*subscriber*" que se una al sistema posteriormente. Permitiendo esto, se consigue la posibilidad de tener una comunicación asíncrona, un "*publisher*" publica en un momento dado, y cuando un "*subscriber*" se una al sistema será capaz de acceder a los datos que se hayan publicado previamente en el sistema, y todo esto es consecuencia de la aplicación de QoS a las distintas entidades que conforman el sistema.

En definitiva, **DDS** está especificado para utilizarse en sistemas distribuidos que tengan la necesidad de intercambiarse información en tiempo real, con independencia de plataforma y lenguaje. Además, provee una gran

escalabilidad y la posibilidad de acceder a datos que se hayan publicado en un momento anterior a la unión de cierto "*subscriber*" al sistema, mediante, como se ha explicado anteriormente, el uso de QoS que permiten ese modelo asíncrono.

2.2.1 Antecedentes de DDS

Se ha considerado relevante mencionar otras alternativas de middleware con características de tiempo real, y para ello se ha decidido comentar **RT**-**CORBA**, que ha sido con diferencia el middleware más utilizado en los últimos años, si bien está cayendo en desuso debido al alza de los sistemas basados en publicación-suscripción y al auge del estándar **DDS**, entre otros.

RT-CORBA:

Se conoce como **RT**-**CORBA** a una versión de **CORBA** (Common Object Request Broker Architecture) que implementa características de tiempo real. Realmente, y como suele ser habitual existen varias implementaciones de la especificación de **CORBA** para tiempo real. **RT**-**CORBA**, la que se trata en esta sección, pertenece a OMG, del mismo modo que **CORBA**.

RT-**CORBA** surge bajo la necesidad de ofrecer capacidad de tiempo real a un middleware tan extendido como **CORBA**. **CORBA** se utiliza normalmente para aplicaciones de propósito general, mientras que **RT**-**CORBA**, sacrifica algunas de las facilidades que ofrece **CORBA** en este sentido para dar al diseñador la posibilidad de utilizar el middleware en aplicaciones críticas, con requisitos fuertes o más suaves, lo que suele ser denominado como "*Hard real-time*" y "*Soft real-time*"

Las características que incluía en su momento son las que ahora se entienden como esenciales, es decir, soporte para lenguajes preparados para tiempo real, y ofrecer una serie de parámetros de calidad de servicio que puedan satisfacer las necesidades de aplicaciones críticas.

La figura siguiente muestra un ejemplo de uso, en una infraestructura con un RTOS como base en el servidor, una máquina Windows en el cliente, y el middleware **RT**-**CORBA** enlazando ambos.

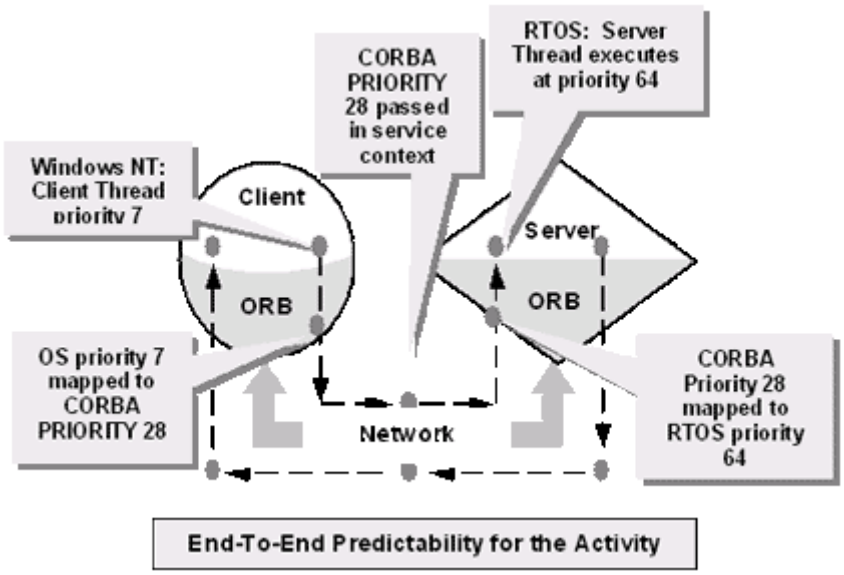


Figura 6: Esquema funcionamiento RT-CORBA

Una de las características importantes de **RT-CORBA**, que permite su versatilidad a la hora de implantarlo en diversos sistemas es que es compatible con la extensión **POSIX** para tiempo real, de modo que la compatibilidad se ve aumentada, una característica que aporta un gran valor.

2.3. Introducción a ICE (Internet Communications Engine)

ICE es un middleware libre, con licencia GPL, desarrollado por la empresa ZeroC y cuya motivación es responder a la complejidad de **CORBA** ofreciendo una funcionalidad similar.

Esta funcionalidad no es otra que la de ofrecer un middleware capaz de proveer un escenario distribuido en el cual se pueda desplegar una aplicación orientada a objetos, consiguiendo que una máquina A acceda a la lógica que implementa una máquina B, por medio de llamadas a métodos remotos pertenecientes a objetos alojados en la máquina B. Esto se consigue haciendo que la máquina A tenga acceso a una interfaz que provea de los métodos que tiene ese objeto alojado en B. Esta interfaz además, tiene que ser genérica, o dicho de otro modo, debe ser nuevamente una interfaz independiente de la plataforma o lenguaje utilizado. De este modo se facilita la escalabilidad además de hacer más versátil el sistema.

Para alcanzar este fin existen muchas tecnologías middleware distintas. Entre los actuales se encuentran **RMI** (Remote Method Invocation) de **Java**, el cual es dependiente de lenguaje (**Java**), **WCF** (Windows Communication Foundation) de Microsoft, el cual es dependiente de plataforma (Windows), **XML-RPC** y **SOAP** (Simple Object Access Protocol), que son dos tecnologías definidas para implementar "*Web Services"*, además de **CORBA** (Common Object Request Broker Architecture) que es posiblemente el más potente, uno de los más extendidos, independiente de plataforma y de lenguaje, y también uno de los más complejos.

ICE surge por tanto con la motivación de cumplir con la funcionalidad que ofrece **CORBA**, siendo independiente de lenguaje y plataforma, pero ofreciendo muchas mayores facilidades a la hora de implementar la solución. Los objetivos principales de **ICE** son:

- Proveer un middleware orientado a objetos, disponible para plataformas heterogéneas.
- Facilitar la utilización de distintos paradigmas, uno de ellos el de

publicación-suscripción, que es en el que se centrará la atención.

- Hacer una plataforma más sencilla, favoreciendo la implantación, uso y aprendizaje de ésta.
- Que la plataforma sea eficiente en cuanto al uso de CPU, memoria y ancho de banda.
- Que la plataforma sea segura por defecto, sin que haya necesidad de añadir parches posteriormente.

A continuación se presentan algunas de las características principales de **ICE**, para comprobar más tarde las correspondientes a **IceStorm**, uno de los módulos que ofrece la plataforma y que es el encargado de ofrecer un modelo de publicación-suscripción, en base al cual se realizara la comparación con **DDS**.

Como se ha comentado, **ICE** se puede utilizar en entornos heterogéneos, esto es posible entre otros motivos, gracias a la posibilidad de hacer que los datos intercambiados sean independientes de plataforma y lenguaje. **ICE** usa un lenguaje, **Slice** (Specification Language for **ICE**) , que hace las funciones de **IDL** para **DDS**.

Otra característica importante es que el middleware **ICE** permite comunicación asíncrona en su modalidad cliente-servidor. El funcionamiento normal, de forma resumida, es que un cliente utiliza su "*proxy*" (proxy es la terminología propia de **ICE**, se asemeja a un "*stub*" de **CORBA** o **RMI**), para realizar una llamada a un método remoto que se encuentra en el servidor, como cualquier llamada normal, se pueden incluir ciertos parámetros, y el hilo del cliente queda bloqueado hasta que el servidor termina de ejecutar ese método y devuelve una respuesta. Pues bien, en el modo de comunicación asíncrono de **ICE**, éste, en la llamada al método remoto, incluye un parámetro de "*callback*", que permite que el hilo vuelva directamente a la ejecución normal, sin quedarse bloqueado, recogiendo la salida del método cuando se le notifique (por medio de esa función de "*callback*"). Esto es gracias al parámetro de "*callback*" que incluye en la llamada.

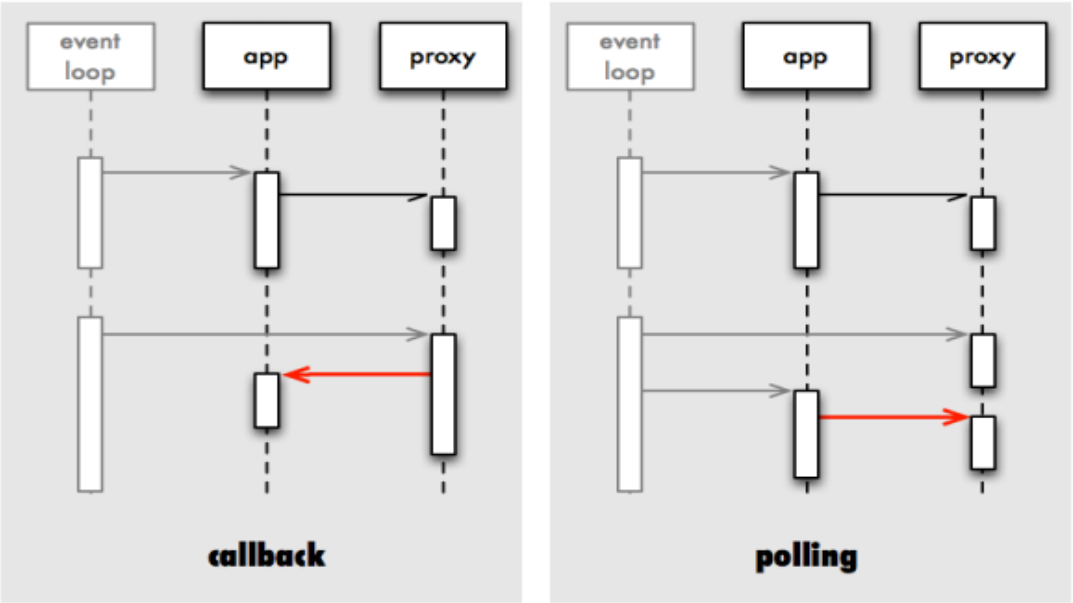


Figura 7: Ilustración comparando callback con polling.

Un punto clave en la tecnología **ICE** son los servicios que provee la plataforma, estos servicios son, **Freeze**, **IceGrid**, **IceBox**, **IceStorm**, **IcePatch2** y **Glacier2**. Cada uno de estos servicios es un añadido al núcleo central de **ICE** y provee a éste de nuevas funcionalidades que en general suponen una mayor comodidad para el desarrollador, además de la posibilidad de acceder a nuevas posibilidades que el núcleo por sí mismo no ofrece.

Servicio	Descripción
IceUtil	Utilidades para manipulación de hilos.(Solo **C++**)
IceBox	Servidor de aplicaciones especifico para **ICE**
IceGrid	Servicio que provee de facilidades para implementar y activar arquitecturas grid.
Freeze	Servicio que dota de persistencia a ciertos componentes del sistema.
FreezeScript	Herramientas de administración de Freeze
IceSSL	Provee de cifrado, autenticación e integridad de mensajes por medio de SSL.
Glacier	Servicio que provee seguridad y facilita la implementación de sistemas distribuidos con firewalls.
IceStorm	Provee publicación-suscripción y “topic federation” al sistema
IcePatch	Provee de un sistema de control de versiones.

Concretamente en nuestro caso, como se comentó anteriormente, se centrará la atención en el servicio **IceStorm**, puesto que es el que dota al núcleo **ICE** de la posibilidad de implantar un sistema de publicación-suscripción, es decir, el paradigma que usa **DDS**, y es por esto mismo que se atenderán especialmente las características principales de este servicio, para usar esta propiedad como eje en la comparación que se realizará.

2.3.1 Introducción al servicio IceStorm de ICE

IceStorm es un servicio cuya funcionalidad reside en dotar al núcleo **ICE** de la posibilidad de realizar su tarea en un paradigma de publicación-suscripción. Esto supone que el sistema que implemente el servicio **IceStorm** tendrá todas las ventajas y posibilidades que ofrece un paradigma de publicación-suscripción, como se comentó en **DDS**.

Dejando de lado las características derivadas de emplear un paradigma de publicación-suscripción, que fueron comentadas en la sección anterior, se verán algunas características propias de la implementación que utiliza **ICE**, que son las descritas a continuación.

En **ICE** se provee un modo de "alta disponibilidad", (*high availability*, a partir de ahora HA), el cual cumple con la función de dotar al sistema de cierta redundancia ante fallos. En este modo existen varias réplicas de cada entidad en el sistema, de las cuales es una la que actúa, si ésta falla, la responsabilidad pasa a la siguiente. La elección de éstas réplicas y el algoritmo que decide qué réplica es la que mantiene el control, es el algoritmo García-Molina, llamado "Invitation Election Algorithm".

Otra peculiaridad de **ICE** con respecto al paradigma de publicación-suscripción (aunque **DDS** implementa el mismo concepto como se verá en la comparación), es lo que denominan con el concepto de "*topic federation*". Consiste en crear un link unidireccional entre los tópicos que se publican, que puedan tener cierta relación para un "*subscriber*". Este link por tanto une un "*topic*" A con otro cierto número de ellos, de este modo, el "*subscriber*" que acceda al "*topic*" A, (que esté suscrito a ese "*topic"*), accederá también a los nodos enlazados desde A. Los mensajes sólo se propagan hacia un "*topic*", nunca más de uno, para evitar bucles e inconsistencias que podría ocasionar no limitar el número de links. Además se incluye también el concepto de coste en el lado de los "*publisher*". Estos costes se asocian al mensaje publicado, indicando en última instancia hacia qué tópicos será enviado. Se consigue debido a que los links entre tópicos también poseen un coste, que debe ser menor que el coste del mensaje publicado, para que éste pueda ser reenviado.

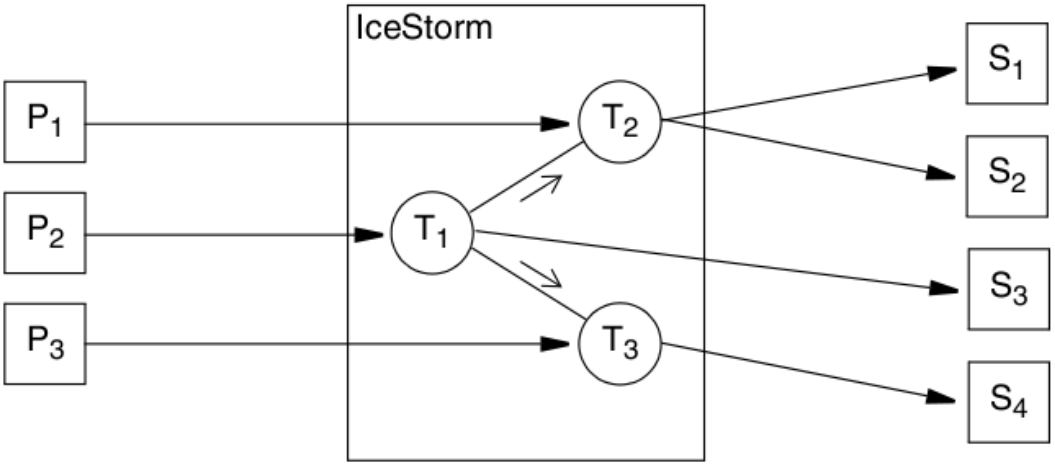


Figura 8: Ilustración que muestra el funcionamiento de "federation"

Al igual que **DDS**, **ICE** también especifica unos parámetros de calidad de servicio QoS, aunque como se comprobará en la comparación, no persiguen los mismos objetivos que en el caso de **DDS**. En el caso de **ICE** éstos parámetros se definen del lado del "*subscriber*", y son básicamente dos. "*Reliability*" y "*Retry Count*", los cuales sirven para garantizar la entrega ordenada ("*reliability*"), e indicar el número de reintentos necesarios antes de eliminar un "*subscriber*" del sistema ("*retry count*"). Este último aplica a situaciones en las cuales hay fallos en la comunicación. Por defecto, **ICE** eliminará el suscriptor que no sea capaz de acceder al mensaje que está publicando, pero con éste QoS puede configurarse este comportamiento.

Por último, y para finalizar la introducción a **IceStorm** conviene observar los tipos de entrega de mensajes de los que dispone.

Define cinco modos:

Twoway ("dos sentidos"): Básicamente el "*publisher*" espera una notificación del "*subscriber*" (lo que se podría entender por un ACK) del último mensaje enviado. Viene a ser el típico mecanismo de llamada y espera.

Oneway ("un sentido"): Al contrario que el anterior, no existe notificación por parte

del "*subscriber*", por tanto es menos fiable pero mucho más eficiente. Este modo unido al uso de UDP supone un mecanismo en el cual el Publisher se desentiende totalmente de los datos una vez los ha enviado.

Batch oneway ("lote en un sentido"): El servicio guarda una pila de mensajes, y los manda de una vez en vez de ir mandando mensaje a mensaje. Mejora el rendimiento en la red, pero aumenta la latencia, ya que el "*subscriber*" se encuentra con una pila de mensajes que procesar a la vez. Si los mensajes son muy pequeños, y se tienen una gran cantidad de publicadores y suscriptores en nuestra red, puede ser muy conveniente, puesto que aun yendo en detrimento de la capacidad de cómputo, una red poco eficiente causaría mayores problemas a más publicadores.

Datagram ("datagrama"): Se envían los mensajes como en el protocolo UDP, no hay garantía de entrega y mucho menos de entrega ordenada, además no se notifican las pérdidas, etc. Si un "*subscriber*" desaparece del sistema sin avisar previamente a éste, el "*publisher*" seguirá enviando mensajes al servicio indefinidamente.

Batch Datagram ("lote datagramas"): Los mensajes se envían como en el caso de "*Batch oneway*", pero con las consideraciones del modo "*Datagram*". Por esto también se tendrá especialmente en cuenta en las mismas situaciones que el anterior.

2.4. Comparación de RTI DDS y ZeroC ICE

Tras la introducción que se ha realizado de **DDS** y **ICE**, en esta sección se va a realizar la comparativa entre estas dos tecnologías, la implementación de **DDS** de RTI y la tecnología **ICE** apoyada en el servicio **IceStorm**. Como se ha comentado y se desprende de las anteriores introducciones, se tienen dos tecnologías cuya aplicación en términos generales difiere bastante. Por ello para realizar esta comparación se va a prestar atención a la aplicación general de cada una, para ir después viendo características más concretas de cada tecnología. De este modo al final de la comparación se tendrá una imagen global de ambas tecnologías y estarán claros los parecidos y diferencias. Además, más tarde se va a introducir **ICE**-**E** (**Embedded ICE**) para comprobar también la aplicabilidad de este middleware a sistemas embarcados.

Por ello el índice que seguirá la comparación será el siguiente:

- Aplicaciones potenciales de cada tecnología.
  - Protocolos.
  - Tipos de comunicación.
  - Parámetros de calidad de servicio QoS.
  - Lenguajes de definición de tipos independientes de plataforma.
- Tecnologías aplicadas a sistemas embarcados.
  - Introducción a **ICE**-**E** y enfoque de la tecnología.
  - Introducción a **DDS** en sistemas embarcados y enfoque de la tecnología.
  - Justificación de los usos de ambas plataformas.
- Conclusión.

2.4.1 Aplicaciones Potenciales de cada Tecnología

Como ya se ha comentado, **ICE** surge como respuesta a **CORBA**, y es por tanto una tecnología que ofrece unas posibilidades muy similares a éste, además de algunas otras que puede implementar con la ayuda de los servicios comentados, como es el caso de **IceStorm**, que le da la posibilidad de montar un sistema de publicación-suscripción.

ICE puede por tanto usarse para desarrollar sistemas distribuidos orientados a objetos, debido a que permite llamadas a métodos remotos. Para una aplicación determinada puede no ser suficiente desplegar ésta en un solo nodo. Con la ayuda de **ICE** puede desplegarla en X nodos, los cuales implementarán **ICE**, de modo que cualquier nodo pueda acceder a la lógica que implementa otro nodo cualquiera o a la suya propia. Esto proporciona facilidad a la hora de repartir tareas en sistemas distribuidos.

Además todos los datos, sean llamadas a métodos, funciones o respuestas de éstos que viajan por la red, se encapsulan utilizando un lenguaje independiente de plataforma, el cual se ha comentado en la introducción, lo que dota de mucha versatilidad a los sistemas que utilizan **ICE**, además de incrementar su capacidad de escalabilidad. Se puede disponer de un sistema distribuido legado, para el que sea necesario desarrollar un módulo encargado de hacer algo muy concreto, como podría ser una interfaz web sobre la que puedan realizarse un cierto número de tareas. Es totalmente viable plantear la posibilidad de utilizar **ICE**, de modo que, desde una interfaz web, se permita acceder a cierto número de funcionalidades, por ejemplo. Estas funcionalidades en realidad podrían estar implementadas en distintos nodos, pero para el usuario sería totalmente transparente, puesto que el middleware **ICE** enlazaría con el resto de nodos realizando las llamadas correspondientes y devolviendo el resultado.

ICE consigue convertir la llamada, ejecución y respuesta de cierta función en una caja negra, siendo independiente y transparente al usuario donde se albergue esa caja negra. Exactamente del mismo modo que hace **CORBA**. Esto ofrece una ligera idea de las aplicaciones para las cuales **ICE** puede ser una tecnología apta.

Por otro lado se dispone **DDS**, que como ya se ha dicho, se ocupa de

facilitar el despliegue de sistemas distribuidos en tiempo real.

DDS nace enfocado a otro campo distinto, primero porque no es un middleware orientado a objetos, es decir, no se pueden realizar llamadas a funciones remotas, más bien es un sistema de paso de mensajes bajo el paradigma de publicación-suscripción. Con **DDS** se posee la misma facilidad que en **ICE** para escalar sistemas, puesto que del mismo modo que éste, **DDS** utiliza un lenguaje que facilita el intercambio de mensajes aunque éstos estén escritos con distintos lenguajes o sobre distintas tecnologías.

El rango de acciones que permite realizar el middleware **DDS** parece menor que el que permite realizar **ICE**. Con **DDS** sólo se puede pasar información y con **ICE** además de esto, realizar llamadas remotas. La diferencia reside en que **DDS** se especializa en la tarea para la cual ha sido desarrollado, además de ofrecer ese servicio en tiempo real.

Con **ICE** nunca se habla de tiempo real, se tiene un middleware de "propósito general", con el que es posible realizar un amplio rango de aplicaciones. Con **DDS** se tiene un middleware centrado en los datos, y especializado para sistemas en tiempo real. Puede afirmarse que **DDS** es mucho más específico que **ICE**, centrándose en un ámbito de la computación distribuida y ofreciendo parámetros de calidad de servicio, que junto a un protocolo, RTPS, que es el que sustenta todo el sistema, son los que mantienen el requisito de tiempo real.

Por todo esto, a priori parecerían dos tecnologías difícilmente comparables, ya que como se aprecia sus ámbitos de aplicación son bastante distintos. No obstante, como se comentó anteriormente **ICE** ofrece un servicio llamado **IceStorm** que permite desarrollar bajo el paradigma de publicación-suscripción. Será éste el punto de partida que se tomará para enfocar los siguientes puntos de la comparación, centrando ésta específicamente en las facilidades que pueda ofrecer una tecnología concreta en ciertos puntos.

2.4.1.1 Protocolos

Para el caso concreto de middleware de tiempo real, como es **DDS**, el protocolo que le de soporte es una pieza muy importante de la arquitectura, por ello, RTI **DDS** se apoya en una capa inferior que es el protocolo RTPS (*Real-Time Publish-Subscribe*), estándar del OMG, que ofrece a la capa superior (la capa de middleware) servicios para facilitar los requisitos de tiempo real. No obstante en general **DDS** puede funcionar sobre UDP o sobre memoria compartida sin una necesidad obligada de la capa RTPS.

ICE funciona tanto sobre TCP, como UDP, como SSL si fuera necesario, es decir, funciona sobre los protocolos estándar de transporte de la pila de protocolos de TCP/IP. De estos protocolos, comentar simplemente que TCP introduce mayor sobrecarga en la red, pero se consigue una entrega ordenada de paquetes y se asegura un control de flujo, así como fiabilidad en la entrega. Sin embargo, en este caso, no por usar TCP se garantiza que en el nivel de aplicación (**ICE** en este caso), los mensajes lleguen ordenados, puesto que puede haber distintos "publisher" publicando mensajes a la vez, por ello es necesario utilizar el QoS *Reliability* que ofrece **IceStorm**, si quiere asegurarse la entrega ordenada. En el caso de UDP, es el protocolo usado en varios de los modos de entrega de mensajes que ofrece **IceStorm**, como se ha visto anteriormente, más concretamente de los modos *Datagram* y *Batch Datagram*. UDP disminuye drásticamente la carga en la red, pero no aporta fiabilidad de entrega ni un posible control de flujo, teniendo que ser todo esto controlado desde la aplicación si se precisase.

Para terminar este punto se verán algunas características de la capa RTPS para comprender las ventajas con respecto a utilizar directamente UDP.

Primero hay que decir que una de las motivaciones del desarrollo de RTPS es aumentar la escalabilidad entre sistemas que implementen **DDS**, sea de la empresa que sea. Es decir RTPS provee de una capa común a todas las implementaciones **DDS** que quieran hacer uso de ellas, de modo que aparte de tener plataforma y lenguaje independiente, se posee así mismo la posibilidad de desarrollar en dos implementaciones distintas de **DDS** (siempre que se basen en el estándar). Con esto las posibilidades de escalabilidad de un sistema aumentan aún más.

En **DDS** se persigue el objetivo de ofrecer un sistema escalable, tolerante a fallos (que no haya puntos centralizados en el sistema) y en el cual las entidades estén muy desacopladas. Todas estas características deben ser respetadas por el protocolo que lo sostiene, y RTPS lo hace. Se presentan a continuación algunas de las características del protocolo en cuestión.

- Proveer propiedades de calidad de servicio para posibilitar entregas de mensajes "*best-effort*" o fiables, y hacer todo esto sobre redes IP.

- Tolerancia a fallos, permitiendo crear redes que no posean un punto único de fallo, es decir, no existen por ejemplo servidores de nombres donde tengan que registrarse las entidades que entran nuevas al sistema ni nada similar.

- Extensibilidad, compatibilidad e interoperabilidad. El protocolo está diseñado para poder cumplir con estos requisitos.

- Soporte "*plug n play*". Significa que las entidades (*publishers y subscribers*) pueden entrar o dejar el sistema en cualquier momento sin depender de un servidor de nombres y sin tener que reconfigurarse cada vez que entran o salen de éste. El protocolo permite realizar estas actividades haciendo que las nuevas entidades sean automáticamente descubiertas.

- RTPS es suficientemente configurable como para permitir realizar un balance entre fiabilidad y entrega a tiempo para cada entrega de datos de forma independiente.

- Es un protocolo muy modular, permitiendo que algunos dispositivos implementen sólo una parte de éste, y que siga siendo posible que funcionen en el sistema. Ésta es una característica muy importante a la hora de tratar **DDS** con sistemas embarcados, pero se verá más adelante.

- Como se ha mencionado tantas veces, escalabilidad. El protocolo supone un punto común a las distintas implementaciones **DDS** basadas en el estándar del OMG.

Por último, indicar que la capa RTPS suele implementarse sobre UDP, teniendo en realidad también soporte para redes multicast. Si se requiriese, RTPS es capaz de dotar de fiabilidad e incluso de estado a la comunicación, sin preocuparse por que el nivel de transporte no lo haga.

Esto hace que sea posible desplegar RTPS sobre un mayor rango de protocolos de transporte.

RTPS necesita del nivel de transporte los siguientes requisitos:

- Que la capa de transporte pueda dirigirse a una dirección de red (una IP por ejemplo).
- Que la capa de transporte pueda dirigirse a un puerto concreto (un socket o una dirección de la memoria compartida.)
- Que la capa de transporte sea capaz de enviar un datagrama al par dirección/puerto anteriores.
- Del mismo modo, la capa de transporte debe ser capaz de recibir un datagrama.
- Que la capa de transporte se preocupe de descartar aquellos mensajes que estén corruptos o incompletos.
- Que la capa de transporte tenga algún medio (por ejemplo un campo), para conocer la longitud de los mensajes que ha recibido.

Como se ve, RTPS es un protocolo muy apropiado para **DDS**, puesto que da servicio a muchas de las características que ofrece éste. Especialmente tiene una gran relevancia a la hora de dar soporte a varios de los QoS que utiliza **DDS**. Muchos de estos QoS requieren controles muy estrictos de tiempo, lo cual sería muy difícil llevar a cabo sin una capa inferior que provea al servicio de las herramientas necesarias. Por ello RTPS es una capa muy importante para **DDS**, conformando junto a ésta el verdadero middleware en tiempo real.

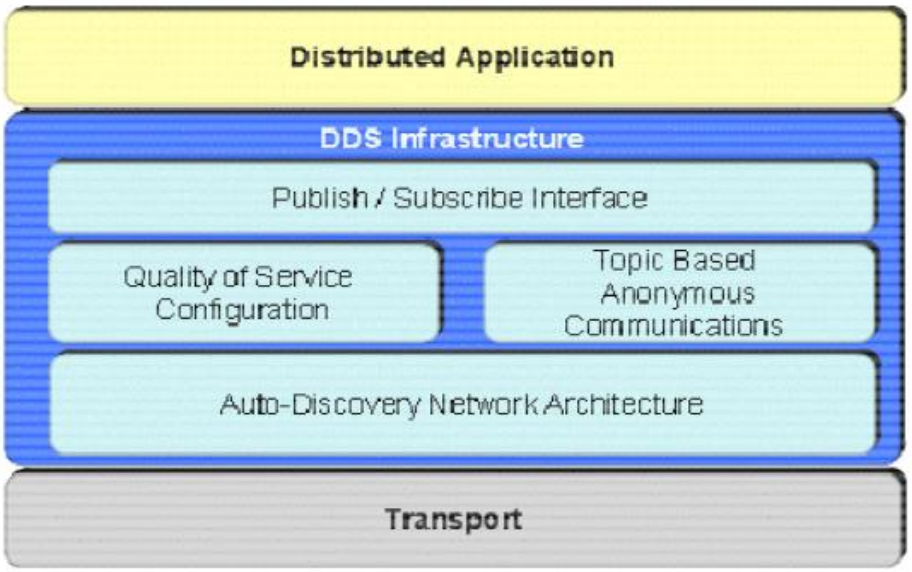


Figura 9: Niveles de la infraestructura DDS, con la capa RTPS.

2.4.1.2 Tipos de comunicación

Tras revisar los protocolos sobre los que funcionan cada una de las tecnologías, se centra la atención en el modo de comunicación de éstas. **ICE** como se ha comentado anteriormente, se enfoca a un rango de aplicaciones más amplio, y por ello define más modos. *Twoway*, *Oneway*, *Batch Oneway*, *Datagram* y *Batch Datagram*, y estos modos se aplican al paradigma de publicación-suscripción si se utiliza el servicio **IceStorm**. **DDS** es un sistema de publicación-suscripción exclusivo y no posee tantas configuraciones posibles para la comunicación. Es lógico, **DDS** se centra en un campo concreto y posee el tipo de comunicación necesario para que ésta se realice correctamente, por otro lado, **ICE** tiene que ser más versátil porque las aplicaciones a las que va a ser dirigido pueden ser de naturalezas muy distintas.

Un punto distinguible entre ambas tecnologías es el acceso a la comunicación, o dicho de otro modo, los pasos necesarios para unirse al sistema y poder comunicarse.

ICE puede hacerlo de dos modos. En una comunicación cliente-servidor puede conocer la dirección/puerto del servidor destino y enviar directamente la

información, o puede también valerse de un "*Location Service*", que viene a hacer la funcionalidad de DNS. Si se quiere utilizar el servicio **IceStorm**, es necesario realizar una serie de configuraciones previas para que se utilice este servicio, así como asignar a los clientes un proxy a través del cual se comunicarán con el sistema.

En **DDS** en cambio, no hay que realizar configuraciones previas, y cuando se precisa añadir un nuevo cliente al sistema, sólo se le tiene que indicar a éste el dominio al cual va a unirse, puesto que pueden coexistir varias comunicaciones entre "*publishers*" y "*susbscribers*" en una misma red. **DDS** implementa un servicio de auto-descubrimiento que realiza los pasos necesarios para que el cliente pueda funcionar en el sistema automáticamente. Esta característica como se ha comentado antes, aporta escalabilidad al sistema, comodidad al desarrollador, y versatilidad a la tecnología, facilitando la portabilidad de ésta a distintos sistemas.

2.4.1.3 Parámetros de calidad de Servicio. QoS

Quizá el punto donde más diferencias existen entre ambas tecnologías, y el que marca la especialización de **DDS** en concreto, sea el de los parámetros de calidad de servicio (QoS).

IceStorm implementa dos parámetros distintos, *Reliability* y *Retry Count.* El primero toma los valores *ordered* o el valor por defecto, que es *not ordered.* Su cometido, en el caso de definir *ordered*, es hacer que **IceStorm** envíe los mensajes a los suscriptores en el mismo orden que le llegaron a él. En ningún momento se comprueba en qué instante salieron de sus respectivos publicadores, simplemente en qué orden llegan al servicio, y éste los envía en el mismo orden en que le hayan llegado. Si se especifica *not ordered*, **IceStorm** envía los mensajes según los recibe.

El segundo se encarga de definir cuándo un subscriber debe ser eliminado del sistema. **IceStorm** puede devolver dos excepciones tras las cuales el subscriber se elimina automáticamente, son aquellas producidas cuando el subscriber no existe, o no está registrado. Para el resto de excepciones es posible definir cuando un subscriber debe ser eliminado. El QoS *Retry Count* toma los siguientes valores: *-1*, *0*, o *un valor positivo*. Con -1 nunca se elimina. Con 0 se elimina en el primer error ocurrido, y con un valor positivo, éste va decrementándose

en cada error, hasta llegar a 0. Este QoS puede ser útil cuando hay, por ejemplo, cortes intermitentes de la conexión.

En el caso de **DDS** existe un número muy grande de QoS, por ello sólo se hablará de aquellos que aportan características que facilitan los requisitos de tiempo real, ya que son los que marcan la diferencia entre las dos tecnologías.

Deadline ("línea limite") indica básicamente cada cuanto tiempo (como mínimo) tiene que publicar un "*publisher*", y cada cuanto tiempo (como máximo) tiene que leer datos un "*subscriber"*. De este modo se puede utilizar este QoS para determinar cada cuanto tiempo se debe publicar, así como cada cuanto tiempo deben leerse datos. Es una gran ventaja para tareas que dan datos de forma periódica, ya que se pueden controlar perfectamente los márgenes de tiempo.

Latency Budget ("ajuste latencia") es un QoS usado para ajustar en la publicación los tiempos, de tal modo que éstos coincidan con la máxima latencia permitida por un "*subscriber*".

Lifespan ("duracion") determina el tiempo máximo que una muestra puede estar en el sistema. Con un valor determinado para este QoS, una muestra es eliminada totalmente del sistema si no ha sido procesada pasado ese tiempo. Hay ciertos datos que sólo son válidos en un tiempo, y pasado ese tiempo, pierden su utilidad, pues con este QoS se puede definir cuando un dato concreto ya no es válido y eliminarlo.

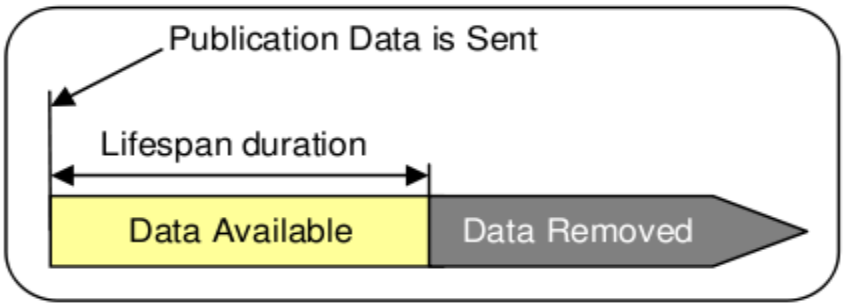


Figura 10: Qos LifeSpan.

Time-Based Filter ("filtro basado en tiempo") es un QoS que especifica cada cuanto tiempo un "*subscriber*" quiere recibir información. Cuando este QoS tiene un valor asociado, un "*subscriber*" solo leerá cada ese tiempo en

concreto, y si hay información que llega antes, la ignorará. Este QoS permite ahorrar los recursos empleados por un sistema si este sistema no necesita todas las muestras que lleguen sino solo algunas cada cierto tiempo.

Además de estos QoS merecen especial atención los siguientes:

Destination Order ("orden") permite indicar al middleware si se desea recibir las muestras de datos en orden de publicación o en orden de llegada. Este parámetro, al contrario del *Reliability* de **ICE**, se basa en marcas de tiempo. Una muestra lleva asociada una marca de tiempo al publicarse, y se le asigna otra cuando llega a su destino. Con este QoS se permite definir si se quiere recibir las muestras atendiendo a la marca de tiempo de origen (ordenadas) o a la de destino (según vayan llegando, sin importar el orden). De esta forma, el mecanismo es mucho más fino y exacto que el correspondiente a **ICE**.

Durability y ***History ("persistencia e historia")*** son dos QoS distintos, pero que en conjunto permiten especificar cuantas muestras publicadas deben ser mantenidas por el sistema, si no han sido entregadas a su subscriber correspondiente, y cómo deben mantenerse. *Durability* permite definir si no es preciso mantenerlas, tomando el valor *Volatile*, si es necesario mantenerlas en memoria (tomando el valor *Transient*), o si precisan ser guardadas en memoria persistente (tomando el valor *persistent*). De este modo un subscriber puede unirse a un sistema que lleve funcionando cierto tiempo y recibir las muestras que se publicaron en este durante el tiempo que no estuvo. *History* define cuantas muestras debe guardar el sistema si se especifica en *Durability* el valor *transient* o *persistent*. Junto a estos dos QoS se encuentra otro nuevo, *Resource Limits* que limita la memoria que puede utilizar la infraestructura **DDS**, de modo que también está limitando la cantidad de muestras que es posible mantener en el sistema. Este QoS a su vez, y del mismo modo que *Time-Based Filter*, permiten en conjunto limitar los recursos que **DDS** utiliza, ofreciendo un modo de configurar el sistema para dispositivos limitados, como pueden ser ciertos sistemas embarcados.

Ownership y ***Ownership strength ("propiedad" y "fuerza de propiedad")*** son dos QoS que usados en conjunto permiten dotar al sistema de redundancia ante errores. La idea es que se tengan, por ejemplo, dos "*publishers*" ofreciendo las mismas muestras de datos a cierto "*subscriber*". El "*subscriber*" sólo

tomará las muestras de uno de los dos "*publishers*", (aquel con mayor valor en *ownership strength*), pero si éste falla, entonces podrá seguir tomando las muestras del otro "*publisher*".

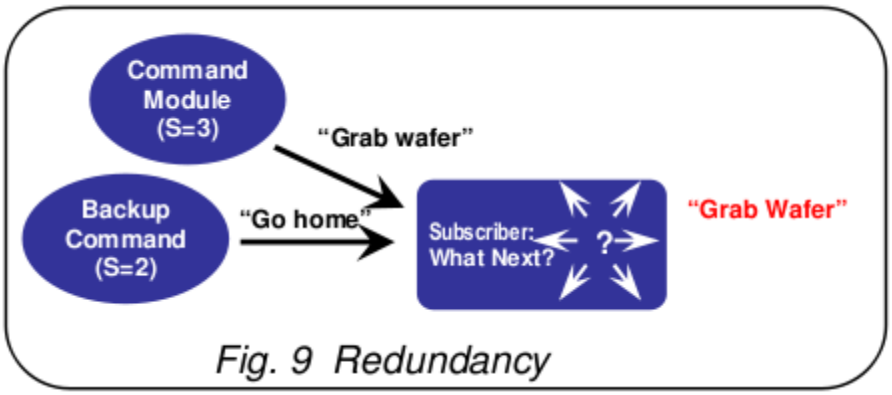


Figura 11: QoS Strength y Ownership Strength

Reliability ("confiabilidad") es el QoS encargado de configurar cómo se va a realizar la entrega de los datos. Esta puede ser *reliable*, o *best-effort*. Con *best-effort*, no hay retransmisiones, con *reliable* en cambio la infraestructura **DDS** se asegura de que todos los datos lleguen a su destino.

Como se observa, **DDS** ofrece muchos más parámetros que **ICE**, proveyendo al middleware de una mayor versatilidad de configuración en cuanto a los requisitos de la comunicación, y especialmente en cuanto a requisitos de tiempo real, donde **DDS** ofrece varios QoS que aportan una gran granularidad.

Como se ha comentado al principio de esta sección, es aquí donde se comprueba la verdadera diferencia entre **DDS** y **ICE**, y donde se aprecia el hecho de que **DDS** sea un middleware para sistemas en tiempo real. Esto se apoya no solo en los QoS que se han comentado sino en el soporte que la capa RTPS ofrece a **DDS** para cumplir con estos requisitos.

2.4.1.4 Lenguajes de definición de tipos

En esta sección se tratará de comprobar las diferencias importantes entre **IDL** (Interface Definition Language), el lenguaje utilizado por **DDS** y **Slice** (Specification Language for **ICE**), el que usa **ICE**.

Es importante comentar el hecho de que **IDL** no es un lenguaje usado exclusivamente por **DDS**, sino que **CORBA** es una de las tecnologías que lo usan. Por ello tanto en **Slice** como en **IDL** es posible definir objetos y enviarlos, así como excepciones etc. No obstante, en nuestro caso concreto se verá que para **DDS** no son necesarias tales definiciones. Por esto se va a enfocar la comparación a lo más básico, la definición de tipos simples, estructuras, pero en ningún caso ninguna estructura que dependa de la orientación a objetos, porque como ya se ha comentado, **DDS** solo pasa mensajes y no realiza llamadas a métodos.

El objetivo de ambos lenguajes no es más que proveer de una interfaz común a los distintos lenguajes de programación que pueden usarse para desarrollar el sistema, de modo que si se tiene un cliente y un servidor desarrollados en dos lenguajes diferentes, no haya ningún problema a la hora de realizar la comunicación. Es decir, estos lenguajes permiten que ambas tecnologías sean independientes del lenguaje, aumentando la escalabilidad del sistema como se ha comentado anteriormente. Actualmente **Slice** permite el mapeo a los siguientes lenguajes: ***C++***, ***Java***, ***C#***, ***Python***, ***PHP*** y ***Ruby***, mientras que **IDL** permite mapeo a los siguientes: ***C***, ***C++***, ***Java***, ***Smalltalk***, ***COBOL***, ***Ada***, ***Lisp***, ***PL/1***, ***Python***.

Como diferencias de tipos entre **Slice** e **IDL**, se encuentran las siguientes:

IDL no tiene soporte para mapear diccionarios, por lo que enviar una *hash-table* de **Java** por ejemplo, sería un proceso complejo, se tendría que transformar la *hash-table* en una cadena de estructuras por ejemplo, **Slice** por otro lado si permite enviar *hash-tables*, puesto que incluye un tipo diccionario.

Slice permite incluir metadatos en cada mensaje enviado, de modo que la estructura de los datos intercambiados se mantenga igual que lo que haya sido

negociado, pero que exista posibilidad de aportar alguna información adicional. En el caso de **IDL** no existe un mecanismo parecido.

Otro problema que se encuentra en **IDL**, aunque este quizá sea más discutible es que se puede llegar a varias formas de realizar la misma tarea, de modo que es difícil llegar a un modo óptimo, para ahorrar memoria y CPU, en **Slice** por el contrario, todos los mapeos están bastante optimizados y suele existir sólo una forma de realizar la tarea. Esto además aporta facilidad a **Slice** y dificultad a **IDL**.

Se aprecia en general que parece más óptimo el código realizado en **Slice** que el realizado en **IDL**, no obstante, estos lenguajes casan muy bien para cada tecnología usada, y desde luego no es un inconveniente el uso del lenguaje **IDL** para la tecnología **DDS** puesto que nunca se verá la necesidad de definir interfaces de objetos remotos, en cuyo caso las diferencias sí se agrandarían más.

2.4.2 Tecnologías aplicadas a sistemas embarcados

En esta sección, se tratarán las características y facilidades de cada una de las tecnologías anteriores comparándolas para el caso de su aplicabilidad a sistemas embarcados. Para esto es importante primero definir el concepto de sistema embarcado, así como comentar algunas pautas que debería seguir un sistema para ser óptimo para un sistema embarcado. Estos sistemas tienen algunas carencias con respecto a los sistemas de propósito general, y es importante tener en cuenta éstas a la hora de desarrollar.

Además también se introducirán las características propias que añade o que diferencian **ICE**-**E**, la tecnología de ZeroC específica para dispositivos embarcados, de **ICE** la tecnología de la que se ha hablado hasta ahora.

Se tratarán especialmente los campos a los que son principalmente aplicadas ambas tecnologías, siendo **ICE**-**E** enfocada a dispositivos embarcados personales, como puedan ser *PDA's, Smartphones*, etc, y **DDS** enfocándose a aplicaciones más críticas relacionadas con sistemas de aviación o defensa entre otras. Además, en la comparación, se prestará especial atención a las claves que hacen que esto sea así.

A continuación se presentan, como se ha dicho, las introducciones a los sistemas embarcados y a **ICE**-**E**.

2.4.2.1 Sistemas Embarcados

Cuando un sistema informático está interactuando directamente con hardware y otros dispositivos de su entorno, ya sea controlándolos o monitorizándolos, y cuando supone en sí mismo una parte de un sistema de información más grande, entonces hablamos de sistema embarcado [19]. En contraste, encontramos los sistemas de propósito general, diseñados para acaparar un mayor número de tareas de cara al usuario final.

Esta definición sugiere a grandes rasgos algunas características comunes a estos sistemas, puesto que es complicado ofrecer una descripción exacta, teniendo en cuenta la diversidad de éstos. Ejemplos de un sistema embarcado pueden ser tanto un sensor de movimiento, un semáforo, como las señales luminosas de la carretera, y también tendrían que aceptarse como sistemas embarcados aquellos sistemas diseñados específicamente para integrarse en sistemas de aviación civil o militar, así como en cualquier medio de transporte. Por supuesto cada uno de estos sistemas tendrá unos requisitos diferentes y enfocados hacia distintos objetivos.

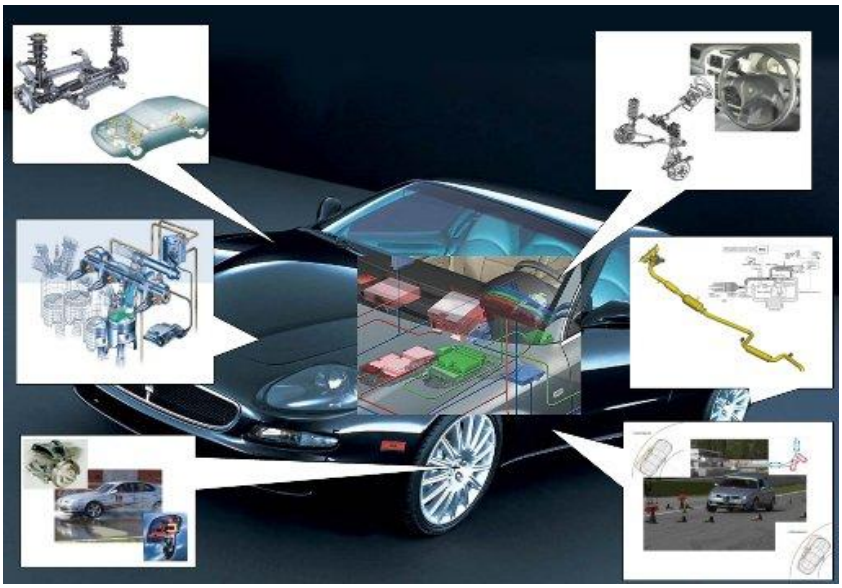


Figura 12: Ilustración de los sistemas electrónicos de un coche.

En muchos casos es necesaria una comunicación entre estos sistemas, para transmitir datos a otro sistema que los procese, o simplemente para compartir éstos. Ejemplos de esto sería una red de sensores en una casa domótica, donde cada uno de los sensores podría ser un sistema embarcado y necesitará comunicar los datos que recoja a otros sensores de la casa para realizar cierta acción bajo ciertas circunstancias.

La motivación del desarrollo de sistemas embarcados es precisamente optimizar un sistema concreto para una tarea específica, de modo que se puede diseñar un sistema mucho más óptimo que un sistema de propósito general como pueda ser un PC. Además en algunos casos no es posible utilizar un sistema de propósito general y es necesario adaptar un sistema embarcado o fabricar uno

nuevo que cumpla con ciertos requisitos de tamaño, fiabilidad ante temperaturas, o condiciones climáticas adversas a las cuales podría ser susceptible de estar expuesto. Casos de esto serían aquellos destinados a colocarse en lo alto de una antena en una montaña, los sistemas colocados a pie de pista en los aeropuertos, o sistemas de señalización ferroviaria, entre muchos otros.

A lo anteriormente descrito es necesario añadir que en un gran número de aplicaciones es necesario que estos sistemas se intercomuniquen entre sí. Anteriormente se comentó el caso de redes de sensores, que de por sí abarca ya un número importante de aplicaciones potenciales. Sistemas de televigilancia, redes de casas domóticas, estaciones de control de montaña, los instrumentos de un vehículo naval o de uno de aviación, y todos aquellos sistemas de computación ubicua que puedan desarrollarse.

Con el avance de la tecnología, los sistemas embarcados se han convertido en una pieza indispensable de una cantidad abrumadora de aplicaciones y sistemas que se utilizan día a día. Desarrollar sistemas cada vez mas óptimos engloba disciplinas como la electrónica, la química, las telecomunicaciones y temas como el control de calidad que en casos concretos como aquellos referidos a sistemas críticos, como puedan ser aviación o sistemas de defensa cobran especial importancia y siguen unas restricciones muy fuertes.

No obstante, aún no se han comentado las necesidades y las restricciones con las cuales se debe desarrollar para sistemas de este tipo, algo que es claramente de una importancia muy alta si el fin es obtener un sistema optimizado. Por ello y teniendo en cuenta que estas restricciones pueden variar mucho de unos sistemas a otros, se comentan a continuación algunos aspectos que son prácticamente comunes en la mayoría de sistemas que puedan encontrarse.

Cuando se quiere desarrollar sobre un sistema embarcado debe tenerse en cuenta las características hardware de éste, como son la arquitectura del sistema, la memoria (tanto volátil como persistente) que se posee, así como las características del microprocesador. Además, es importante tener claros los interfaces de comunicación de los que disponga, si dispone de alguno, y otras características adicionales que pueda implementar como sean las características de sensores o actuadores.

Especial importancia toma tener extremadamente claro el fin y objetivo que se busca, en aras de conseguir un código eficiente, que encaje con las especificaciones tanto de hardware del sistema embarcado como las especificaciones de la aplicación a desarrollar.

Para esto es importante tener claras una serie de cosas:

- El código que se desarrolle debe manejar la memoria del sistema de forma determinista, de modo que no se puedan generar desbordamientos de la memoria del sistema embarcado, lo cual podría dejar inútil el sistema. Por tanto tener muy claro el flujo de ejecución del programa es algo básico.

- Además hay distintas formas de solucionar un problema, en una aplicación desarrollada para un sistema de propósito general, puede que convenga quedarse con la solución que sea más fácil de implementar, buscando la sostenibilidad del proyecto, por ejemplo, pero en sistemas embarcados suele ser habitual optar por la solución que consuma menos ciclos de reloj, menos memoria y que por tanto sea más rápida, aun sacrificando la facilidad a la hora de desarrollar.

- Implementar un código seguro (en cuanto a errores y bloqueos) es también una característica primordial en sistemas de este tipo. Descubrir errores en sistemas grandes, suele ser una tarea complicada, puesto que en general no se poseen los mismos medios que pueden aplicarse a sistemas de propósito general. Por este motivo cobra especial relevancia controlar la concurrencia de los procesos que puedan ejecutarse en el sistema, así como los accesos a recursos compartidos por estos.

- En el caso de implementar comunicaciones entre varios sistemas embarcados, es necesario que el software utilizado se adapte a la red que se va a buscar, y que provea de una solución sencilla sin mucha sobrecarga, para evitar tiempos innecesarios.

Una característica que también puede encontrarse comúnmente en una gran cantidad de aplicaciones es el hecho de que sean necesarias características de tiempo real. Por tanto muchos de estos sistemas embarcados, particularmente aquellos que requieren tareas avanzadas, suelen llevar un sistema operativo en tiempo real, que ha sido previamente testeado para el hardware en concreto que implemente. En muchos casos por tanto, las aplicaciones son desarrolladas sobre

estos sistemas operativos.

En el caso de que se esté tratando con aplicaciones críticas en las que el tiempo real sea una característica determinante, es una ventaja muy grande contar con software específico, middleware, o sistemas operativos que implementen mecanismos para reducir la aleatoriedad de la ejecución de nuestras aplicaciones. Para el caso de las comunicaciones toma especial relevancia por el carácter no determinista de las comunicaciones a través del medio, ya sea cable o radio.

Por todo esto, contar con un middleware que provea facilidades de tiempo real a la hora de realizar las comunicaciones entre sistemas es algo muy deseable. En el caso que está tratándose, tanto **DDS** como **ICE**-**E** tienen características de tiempo real, las cuales serán el eje que se seguirá para realizar una comparación entre ambas y entender por qué cada tecnología se enfoca a las aplicaciones que van a verse.

2.4.2.2 Introducción a ICE-E y enfoque

En esta sección se revisan las diferencias existentes entre **ICE**, el middleware del que se ha estado hablando durante la primera parte de la comparación y **ICE**-**E** la tecnología desarrollada específicamente para sistemas embarcados. Se centra la atención particularmente en las características de tiempo real que introduce **ICE**-E con respecto a **ICE**.

Como se comentó en la sección anterior (2.4.2.1), en una gran cantidad de casos, las aplicaciones que corren sobre sistemas embarcados requieren comportamientos en tiempo real, siendo importante en estos casos tener herramientas que nos permitan configurar ciertos aspectos del software, como los recursos a utilizar, o prioridades de tareas. Por este motivo es que se desarrolla **ICE**-**E**, introduciendo sobre la base de **ICE** funcionalidades de tiempo real, y eliminando de esta base características que no cobran mucho sentido cuando se tratan sistemas embarcados. A continuación se muestran algunos cambios importantes con respecto a lo que se ha contado en la primera parte de la comparación.

Se pueden encontrar los siguientes cambios con respecto a **ICE**:

- En **ICE**-**E** se han suprimido una gran cantidad de características, y hay otras tantas que ahora son opcionales. El objetivo de esto es reducir el tamaño del middleware, siendo por tanto posible embarcarlo en dispositivos limitados.

- **ICE**-**E** sólo soporta (al igual que **ICE**) como modelo de concurrencia, “thread pool concurrency model”, donde el middleware dispone un pool de hilos, de modo que quien lo necesite tenga un hilo preparado previamente. Esta técnica evita el coste en tiempo que supondría la creación y configuración del hilo.

- En cuanto a protocolos, vimos que **ICE** podía colocarse sobre TCP, UDP, o SSL. En el caso de **ICE**-**E** sólo soporta TCP, lo cual inherentemente supone algunos cambios en los modos de transmisión, y significa que algunas de las posibilidades de **ICE** no estarán presentes en **ICE**-**E**.

- En **ICE** existe la opción de realizar comunicaciones asíncronas, a través de funciones de *callback* que se pasaban de cliente a servidor, y que retornaban el valor al cliente cuando el servidor había terminado su tarea, sin necesidad de que el cliente esperase explícitamente a que el servidor terminara su ejecución. Esta característica en **ICE**-**E** ha sido eliminada.

- API's de *streaming* y similares también han sido eliminados de **ICE**-**E**.

- El recolector de basura que era el que se encargaba de eliminar objetos o recursos que ya no estuvieran utilizándose ha sido eliminado. (Para el caso en que fueran asignados por valor). Esta opción puede entenderse como una forma de “obligar” al desarrollador a realizar asignaciones por proxy (lo que fuera del contexto de **ICE** es por referencia), en vez de generar objetos nuevos para cada movimiento, lo cual es, por supuesto, mucho más eficiente.

- En el lenguaje **Slice**, se podían realizar *checksums* de los datos de modo que se comprobase su integridad en destino, en el caso de **ICE**-E se ha eliminado esta posibilidad, de modo que no hay control de errores explícito en **Slice**. Esto reduce la carga de datos que viajan por la red, así como coste computacional de la generación y comprobación de estos *checksums*.

Una característica importante es que **ICE**-**E** es compatible con **ICE** en código, siempre y cuando se mantengan las restricciones anteriormente descritas (y algunas más que su especificidad no se comentan aquí). Esto supone una posibilidad de mantener un sistema en el que convivan nodos con **ICE** y sistemas embarcados con **ICE**-**E**.

Tras comentar un poco las diferencias a nivel general se pasa a comentar las características de tiempo real de **ICE**-**E** que es una de las partes clave que hacen de esta tecnología apta para una gran cantidad de sistemas embarcados.

- Prioridad de hilos. En el Thread API de **ICE**-**E** se incluye un parámetro a la hora de arrancar hilos, este parámetro es la prioridad. Entre la ejecución de varios hilos, tomará mayor importancia, y posiblemente tenga un acceso más rápido a los recursos aquel hilo con la prioridad más alta. Esta técnica permite que se ajuste una prioridad mayor a aquella actividad que sea necesario terminar en un espacio de tiempo menor, pudiendo por tanto favorecer la ejecución de cierta tarea que tenga unos requisitos de tiempo concretos en favor de otras ejecuciones que no tengan tales requisitos. Hay que indicar que el valor de la prioridad debe concordar con la norma POSIX.

- **ICE**-**E** incluye también algunas mejoras de tiempo real para el API de exclusión mutua, Mutex API. No obstante son aspectos que no merece la pena destacar en este punto por la especificidad de los mismos.

- El Timer API incluye también la posibilidad de que los hilos se ejecuten con prioridades.

- Inversión de prioridades. Es un problema que aparece cuando un hilo de una prioridad baja, evita la ejecución de un hilo con una prioridad más alta por poseer un cerrojo. Esta situación es, evidentemente, poco deseable, y **ICE**-**E** provee de una solución para evitarla. La solución consiste en que aparte de los cerrojos normales,

existan otros que sean coherentes con las prioridades de cada hilo. Una vez más el funcionamiento en detalle no es importante para la comparación. Una cuestión a destacar es el hecho de que esta situación podría ocurrir en un sistema distribuido. Varios clientes con una prioridad baja y varios con una alta realizan peticiones sobre un mismo servidor, **ICE**-**E** no trata este problema, y por esto los clientes con prioridad más baja retardarán a aquellos con prioridad más alta.

Además de estas características, **ICE**-**E** también permite una granularidad bastante alta a la hora de configurar la plataforma, decidiendo qué librerías incluir y cuáles no, para conseguir que el tamaño físico del middleware sea el menor posible y utilizar el menor número de recursos posibles.

Tras repasar las características más importantes que diferencian **ICE**-**E** de **ICE**, así como las facilidades de tiempo real que incluye **ICE**-**E**, se va a proceder a ver hacia qué aplicaciones se enfoca **ICE**-**E**, teniendo en cuenta que no se limitan sólo a las que van a comentarse, simplemente se pondrán algunos ejemplos de sistemas en los cuales se está utilizando y se intentara ver el porqué.

Las plataformas oficialmente soportadas por el middleware **ICE**-**E** son: Windows Mobile 6 Professional (arquitectura ARM), Windows XP y Vista, Gumstix Linux Buildroot y Red Hat Enterprise Linux 5.2. En cuanto a soporte para dispositivos móviles, se tiene **ICE** para Android, para iPhone, etc, pero no hay soporte de **ICE**-E para estas plataformas, por tanto sólo se tratarán las comentadas primeramente.

Run-Time Platform	*Development Platform*	*Compiler*
Windows Mobile 6 Professional (ARMV4i)	Windows XP (x86)	Visual C++ 2005 SP1 and 2008 with Smart Device support
Windows XP (x86) and Vista (x86 and x64)	Same as run time	Visual C++ 2005 SP1 and 2008
Gumstix Linux Buildroot revision 1364	Red Hat Enterprise Linux 5.2 (i386)	GCC 4.1.1 cross-compiler
Red Hat Enterprise Linux 5.2 (i386 and x86_64)	Same as run time	GCC 4.1.2

Figura 13: Cuadro de compatibilidad de plataformas de Ice-E

Windows Mobile 6 y Gumstix Linux están enfocados a sistemas embarcados, y serán los que cubran la atención. Estos sistemas suelen encontrarse en SmartPhones, y quizá en algunas PDA's. Windows Mobile es un sistema operativo con soporte para aplicaciones de tiempo real, por lo que el middleware **ICE**-E puede aprovechar sus características sobre éste.

ICE-**E** es una tecnología que sólo permite (actualmente) mapeo de datos en **C++**, lo cual hace que pierda cierta versatilidad, así como escalabilidad con respecto a otras soluciones. La programación es relativamente sencilla, debido a unos API's y una documentación muy completa, con lo que consigue ser una plataforma con la cual se pueden desarrollar aplicaciones no demasiado complejas en un tiempo bastante reducido. Además incorpora algunas características de tiempo real, que si bien no suponen una solución completa de tiempo real, si proveen algunas facilidades a la hora de utilizar características de Windows Mobile por ejemplo. Es una tecnología muy nueva, y quizá por esto las plataformas soportadas sean tan reducidas. No obstante se aprecia que se enfoca hacia dispositivos móviles de uso habitual. A priori no es una tecnología apta para aplicaciones que vayan un poco más allá, como puedan ser aplicaciones en las que se requiera tiempo real estricto, y que sean de un carácter crítico, pero tampoco es difícil pensar que pueda aplicarse a dispositivos que se salgan de los oficialmente soportados, debido a que se apoya en varios sistemas operativos bastante difundidos.

Básicamente, puede llegarse a la conclusión de que es un middleware bastante potente para desarrollar aplicaciones no críticas, sin necesidades especiales de escalabilidad, y cuyos requisitos de tiempo real no sean especialmente amplios, debido a que incluye sólo algunas características, que si bien pueden casar bien con ciertos RTOS como Windows Mobile, no suponen una base fuerte para desarrollar aplicaciones que realmente necesiten requisitos avanzados de tiempo real estricto.

2.4.2.3 Introducción a DDS en SE y enfoque

De **DDS** hay que comenzar diciendo que no tiene un software especial para sistemas embarcados (existe un matiz que se comenta más adelante), el middleware en sí es apto para todo tipo de aplicaciones, lo cual lo hace realmente

versátil, manteniendo la libertad de lenguajes y siendo compatible con un gran número de arquitecturas tanto de propósito general como de sistemas embarcados como se verá más adelante. Esto se hace posible gracias a la potencia que ofrecen los QoS que define, los cuales permiten al desarrollador configurar su aplicación con una granularidad muy alta.

Todos los requisitos que se han comentado secciones atrás sobre las pautas que había que tener en cuenta a la hora de programar sobre sistemas embarcados, son perfectamente alcanzables mediante los QoS que ofrece **DDS**. Hay que indicar también que esto se hace posible gracias al papel que juega el protocolo RTPS para el middleware, el cual ofrece muchos mecanismos para que se realice todo de forma segura y eficiente.

Por esto, se comienza repasando las características generales de **DDS** que hacen que sea apto para sistemas embarcados, tras lo cual se resumen algunos QoS, que aunque fueron descritos anteriormente, se muestra ahora su aplicabilidad enfocada a sistemas embarcados.

DDS es un middleware cuyo tamaño es bastante reducido, el cual se apoya en una capa RTPS que le dota de potentes mecanismos que facilitan y permiten sus características de tiempo real. Además tiene una extensión a RTSJ, es decir, Real-Time **Java**, y unido a esto soporte para los principales RTOS existentes, lo cual es una gran ventaja de cara a aplicaciones críticas que necesiten requisitos realmente estrictos de tiempo real.

Además y como acaba de comentarse, tiene un gran número de parámetros de calidad de servicio que permiten configurar el comportamiento de la aplicación aumentando el grado de determinismo de éstas. A continuación se pasa a comentar estos QoS enfocándolos a las necesidades de sistemas embarcados.

- ***Durability*** **e** ***History***: Se encargan de configurar si las muestras permanecerían en el sistema, y en caso afirmativo, cuantas y de qué manera (en disco o en memoria volátil). Para un sistema distribuido de sistemas embarcados, el cual se encuentra, supóngase, en un entorno tal que la comunicación pueda ser intermitente, y todas las muestras sean necesarias, se hace necesario utilizar estos mecanismos para

mantener las muestras en el sistema. No obstante si un sistema cae por completo, no habría problema en desbordar la memoria del sistema con muestras, puesto que es posible definir exactamente cuántas muestras mantendrá el sistema, de modo que se puede configurar al inicio un número de memoria máximo, lo cual es una opción muy ventajosa.

- ***Lifespan***: Este QoS permite que en una aplicación en la cual las muestras solo tengan validez durante un tiempo determinado, este tiempo sea ajustado, siendo las muestras eliminadas del sistema tras este tiempo, evitando de este modo el consumo de recursos inútiles. Es otra característica de optimización que siempre es recomendable tener en cuenta a la hora del desarrollo de una aplicación para un sistema embarcado.

- ***Ownership* y *Ownership Strength***: Con estos parámetros se consigue dotar al sistema de redundancia ante fallos. Para una aplicación distribuida esto es muy importante, y si la aplicación se basa en sistemas embarcados además, se tiene un modo de configurar en cada sistema qué muestras interesan (las de mayor *ownership strength*), o si se quieren todas (ignorar el valor de *ownership strength*). Proveyendo de otro modo efectivo de controlar los recursos de nuestro sistema

- ***Reader Data Lifecycle ("ciclo de vida del lector")***: Nuevamente se tiene otro QoS que nos proporciona facilidades para liberar recursos cuando estos ya no son necesarios. Un "*subscriber*" será eliminado del sistema, liberando recursos, cuando haya pasado un tiempo desde la última vez que recibió datos por parte de un "*publisher*". Es decir, permite especificar un tiempo límite tras el que si no se han recibido datos, se liberará el recurso.

- ***Resource Limits ("limite de recursos")***: Es uno de los parámetros más importantes a la hora de afrontar el desarrollo de una aplicación en un sistema embarcado. Nos permite configurar de una forma muy fina el número de recursos que utilizará nuestro sistema. Puede especificarse el total de memoria local que utilizará la infraestructura **DDS**. Permite establecer límites para el máximo número de tópicos,

para el máximo número de datos por tópico, o para el máximo número de datos por instancia de cada tópico.

- ***Time-Based Filter***: Nos permite establecer en el "*subscriber*" el tiempo mínimo entre muestra y muestra recibida, evitando de este modo que se produzcan sobrecargas de las colas de recepción, y haciendo determinista la recepción de muestras nuevas. En un sistema en el que se está recibiendo información de varias fuentes, este QoS permite muestrear esa información, en el caso de que no toda sea importante, consiguiendo de esta forma reducir los recursos utilizados.

- ***Writer Data-Lifecycle ("ciclo de vida del escritor")***: El funcionamiento de este parámetro es muy similar al QoS *Reader Data-Lifecycle*. Si un "*publisher*" que era responsable de publicar datos de un tipo se encuentra ante una situación tal que los datos a los cuales está suscrito no son necesarios en el sistema, y está configurado con este parámetro de calidad de servicio, el middleware **DDS** se encargará de eliminarlo, liberando recursos del sistema.

A todas estas características de **DDS** hay que volver a insistir en el fuerte soporte que le dota la capa RTPS, la cual además es altamente configurable, pudiendo el desarrollador elegir qué funcionalidades de ésta estarán presentes en la aplicación, con el fin de disminuir el tamaño físico de la aplicación y el determinismo de la misma.

Tras repasar las características que ofrece **DDS** y que posibilitan su despliegue en sistemas embarcados, se centra ahora la atención en las aplicaciones para las cuales se enfoca y es utilizado en la actualidad.

A diferencia de **ICE**-E las aplicaciones de **DDS** son bastante más amplias y suelen cumplir con unos requisitos mucho más estrictos. Suelen ser aplicaciones para sistemas de defensa, sistemas de aviación tanto civil como militar, control de tráfico aéreo, control de transacciones financieras, o control de procesos industriales entre muchos otros.

DDS es soportado de forma oficial para la gran mayoría de plataformas de propósito general que existen además de para una gran variedad de sistemas embarcados. Tiene soporte para Linux, Solaris, Windows 2000 en adelante, y para la mayoría de sistemas operativos en tiempo real, INTEGRITY, LynxOS, QNX, VxWorks, y Windows Mobile. Además soporta la mayoría de arquitecturas de propósito general, sean, AMD 64, ARM, PowerPC, SPARC, x86. Y concretamente para el caso que nos ocupa que es el de los sistemas embarcados, hace falta comentar que algunos de los fabricantes de RTOS tienen convenios con RTI (la empresa de la distribución **DDS** que se está comparando). Algunos casos son INTEGRITY de Green Hills Software, vxWorks de Wind River, Windows Mobile de Microsoft o LynxOS de LynuxWorks.

Supported Host Platforms	Operating System
AIX®	IBM XLC for AIX v9.0
Linux®	Red Hat® Enterprise Linux® 3.0, 4.0, and 5.0
	Red Hat Linux 8.0 and 9.0
	SUSE Linux Enterprise Server 10.1
Solaris™	on UltraSPARC: Solaris 2.8, 2.9, and 2.10
	on Pentium (x86): Solaris 2.9 and 2.10
Windows®	Windows 2000 with service pack 2 or higher Windows 2003, Windows 2003 x64 Edition Windows Vista® Windows XP Professional, Windows XP Professional x64 Edition

Supported Target Platforms	Operating System	Reference
AIX	IBM XLC for AIX v9.0	Table 1.1 on page 5
INTEGRITY®	INTEGRITY 5.0	Table 1.2 on page 5
Linux	Red Hat Enterprise Linux 3.0, 4.0, and 5.0 Red Hat Linux 8.0 and 9.0 SUSE Linux Enterprise Server 10.1	Table 1.3 on page 5

Supported Target Platforms	Operating System	Reference
LynxOS®	LynxOS 4.0, LynxOS 4.2, and LynxOS-SE 3.0	Table 1.4 on page 7
QNX®	QNX Neutrino® 6.3.0	Table 1.5 on page 7
Solaris	Solaris 2.8, 2.9, and 2.10	Table 1.6 on page 8
VxWorks®	VxWorks 5.4.2, 5.5.1, 6.0 - 6.5	Table 1.7 on page 9
Windows	Windows 2000 with service pack 2 or higher Windows 2003, Windows 2003 x64 Edition Windows CE 6.0 Windows Vista Windows XP Professional, Windows XP Professional x64 Edition	Table 1.8 on page 10

Figura 14: Compatibilidad general de DDS.

A continuación se muestra un ejemplo de las arquitecturas que soporta para el sistema operativo de INTEGRITY, solo a modo de ejemplo, puesto que incluir todas las arquitecturas posibles llevaría demasiado espacio.

INTEGRITY Platforms

Operating System	CPU	IP Stack[1]	RTI Architecture Abbreviation
INTEGRITY 5.0.7	PPC 74XX	InterNiche (GHnet1) TCP/IP stack	ppc7400Inty5.0.7.mvme5100-7400 [2]
		Interpeak TCP/IP stack with multicast	ppc7400Inty5.0.7.mvme5100-7400-ipk
INTEGRITY 5.0.8	PPC74XX	InterNiche (GHnet1) TCP/IP stack	ppc7400Inty5.0.7.mvme5100-7400[2]
INTEGRITY 5.0.9, 5.0.10	PPC74XX	GHnet2 TCP/IP stack	ppc7400Inty5.0.9.mvme5100-7400-ghnet2

Figura 15: Figura especifica de compatibilidad para plataformas Integrity

El párrafo anterior expone de forma general la idea de que **DDS** está ampliamente distribuido, y que está testado frente a RTOS como los descritos. Para aplicaciones embebidas esta compatibilidad va aun más lejos, puesto que **DDS** es apto en muchos casos para sistemas embarcados construidos directamente por empresas de RTOS especializadas en sistemas embarcados, y las cuales llevan consigo un RTOS integrado. De este modo, **DDS** puede integrarse en placas altamente optimizadas para aplicaciones críticas de tiempo real estricto que llevan un RTOS embarcado.

La compatibilidad para sistemas embarcados va aun más lejos. Hay una extensión de **DDS** conocida como "*Safety-Critical Edition*" que cumple con el certificado RTCA/DO-178B (EUROCAE ED-12B), y que está testada contra plataformas Linux, Solaris y vxWorks, y que se supone compatible con INTEGRITY y LynxOS. Este certificado supone la validación del middleware (bajo alguna que otra circunstancia) para utilizarse en aplicaciones críticas.

Puede entenderse esta utilización de **DDS**, debido a la granularidad de los QoS que ofrece, al uso de la capa RTPS que da soporte a muchas funcionalidades de tiempo real, el amplio soporte que tiene con sistemas operativos en tiempo real y con ciertas placas que utilizan estos. La independencia de lenguaje y los certificados DO-178B dotan aun de más posibilidades al middleware.

Además, no se puede pasar por alto algunas extensiones de **DDS** que ofrecen funcionalidades como la de la compatibilidad de **Java** en tiempo real, la edición "*safety-critical*", y otras ediciones como "*Financial Services Edition*" que está preparada concretamente para aplicaciones de transacciones financieras.

Por último, añadir que **DDS** ofrece una compatibilidad muy estable con **CORBA**, uno de los middleware orientados a objetos más extendidos en la actualidad, lo cual aumenta las posibilidades a la hora de implementar una aplicación en la cual sea necesario el uso de **CORBA**.

2.5 Conclusión

Las diferencias entre **DDS** y **ICE** son claras, como se concluyó en la primera parte de la comparación, **ICE** es un middleware orientado a objetos mucho más general, **DDS** un middleware centrado en los datos más específico, **ICE**-E introduce algunas características que permiten que se utilice en ciertos dispositivos, pero **DDS** tiene un enfoque mucho más profundo en el campo de las aplicaciones de tiempo real y los sistemas embarcados en concreto.

Si es necesario desarrollar una aplicación distribuida orientada a objetos, que requiera una huella de memoria mínima, la opción es clara: **ICE**. Si por el contrario ésta se enfoca a aplicaciones distribuidas de tiempo real críticas, la opción es **DDS**. Cuando se habla de aplicaciones distribuidas en las que no prima la orientación a objetos ni los requisitos de tiempo real, se tiene **DDS**, que implementa un paradigma de publicación-suscripción y está **ICE** que es mucho más versátil en este sentido, ofreciendo tanto el paradigma de publicación-suscripción como uno de cliente-servidor tradicional, no obstante en éste caso habría que realizar un estudio profundo de los requisitos de la aplicación a desarrollar y de cómo ambas tecnologías resolverían el problema, atendiendo a puntos concretos de ambas y eligiendo en función de esto, además de tener en cuenta la implementación concreta de la tecnología en el caso de **DDS**.

Si se tratan sistemas embarcados, **ICE**-E es una tecnología nueva que tiene por el momento poco soporte y que está muy enfocada a un sector, en de los dispositivos limitados personales, smartphones, pda's, etc. pero tiene una curva de aprendizaje rápida y está muy bien documentada. Para el resto de aplicaciones embebidas distribuidas, y especialmente, si éstas tienen requisitos de tiempo real y requieren la utilización de un estándar, **DDS** es la mejor opción.

Capítulo 3: Desarrollo de la demo de videovigilancia remota.

3.1 Resumen

En este capítulo se pretenden mostrar las partes de la aplicación que se ha desarrollado, justificando las decisiones tomadas durante el proceso y comparando éstas con otras alternativas existentes. Además se explicarán las partes más relevantes del código de la aplicación, en aras de proveer una visión más fina del funcionamiento de ésta.

Antes de comenzar, es importante recalcar que la aplicación persigue un objetivo meramente didáctico y experimental, como se comentó en el primer capítulo, por lo que muchas decisiones de diseño se toman obviando la eficiencia de la aplicación, y a veces la sencillez, hecho que se encontrará en algunas ocasiones y que se indicará apropiadamente. El objetivo principal del trabajo es esclarecer las posibilidades, facilidad de funcionamiento, y posibilidades de implementación de la tecnología de middleware **ICE**. Este trabajo se desarrolla dentro de un proyecto europeo en el que uno de los objetivos es evaluar **ICE** como middleware base de desarrollo de sus demostradores.

Con este objetivo fijado desde un inicio, se establecen una serie de requisitos, que se incide nuevamente en que no serán óptimos (en la mayoría de casos) desde el punto de vista de la eficiencia de la aplicación, pero sí pretenden ser los más útiles para medir el middleware.

Además, también se detallará en este capítulo la plataforma donde se ha desarrollado y probado esta aplicación, las versiones de las tecnologías utilizadas, y demás consideraciones necesarias, que junto con los anexos que se incluyen al final del documento en el que se detallan las configuraciones e instalaciones específicas de ciertas partes de la aplicación, pretende facilitar la reproducción del entorno en el cual se ha desplegado para poder montarse en cualquier momento.

El aspecto de la aplicación en funcionamiento se muestra en la siguiente fotografía, donde se aprecian los dos nodos y la webcam. En ambos monitores se observan las respectivas aplicaciones, en una interfaz **Java**, en la que se ve lo que está capturando la webcam en ese momento.

Al final del documento se encuentra el anexo 4, en el cual se encuentran

más capturas de la aplicación, para mostrar su aspecto en varios estados.

Figura 16: En la figura, monitor y nodo izquierda (“cliente”), monitor y nodo derecha (servidor). También se observa la webcam conectada al nodo cliente.

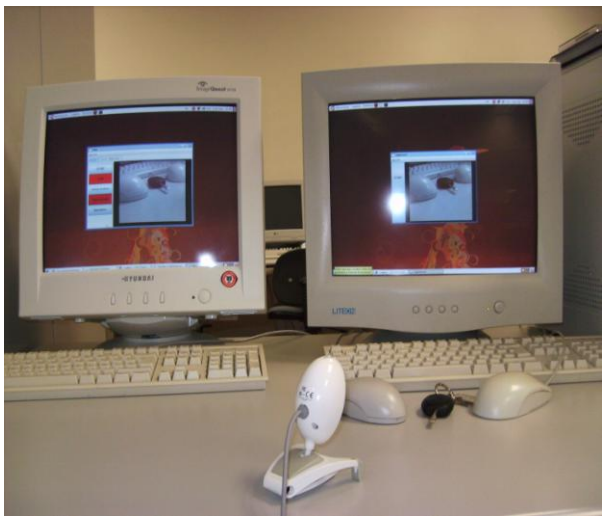


Figura 17: Fotografías del funcionamiento de la aplicación.

En las dos figuras de arriba y en la inmediatamente superior, se aprecia la captura de la webcam en ambos nodos.

Este capítulo por tanto se destina a explicar cómo se ha conseguido la funcionalidad anterior.

3.2 Contexto de la aplicación

La aplicación implementada busca la realización de tres tareas principales, que son: la captura de imágenes desde un dispositivo de captura y su presentación en pantalla, la posibilidad de guardar este vídeo en disco, y la comunicación mediante un middleware específico con otro nodo, de modo que el vídeo capturado en el nodo origen se visualice en la pantalla de un nodo remoto.

Estas tareas se producen en un entorno de publicación-suscripción, implementado por el servicio **IceStorm**, perteneciente a la tecnología **Ice**.

Para este fin se ha implementado una aplicación cliente, que es la encargada de capturar vídeo del dispositivo de captura, guardarlo en disco y enviarlo a otra aplicación servidora, que se encarga simplemente de recibir el vídeo y presentarlo asimismo en pantalla.

Antes de entrar en más detalles, a continuación se detalla el entorno en el cual se ha desarrollado la aplicación. Se ha preparado un entorno distribuido compuesto por dos ordenadores en una red local (**LAN**), interconectados por un switch, modelo **TRENDNet TE100-S24 10/100 Mbps NWay**. Ambos sistemas poseen un procesador **P-IV**, con una frecuencia de micro de **1.8Ghz**, y en ambos casos la memoria RAM es de **256 MB**. En ambos sistemas se ha utilizado como sistema operativo **Ubuntu 8.04 (Hardy).** La versión del kernel es la **2.6.24**. Se utiliza **Java** tanto en la parte cliente y parte servidora, la versión utilizada en este caso es la **JDK 1.6**. Además, la parte de comunicación relativa a **IceStorm** corresponde a la versión del middleware **ICE 3.3.1**. Las partes concretas que implementan el "*publisher*" y el "*subscriber*" se realizan en **C++**, compiladas y ensambladas con **GCC versión 4.2.4**.

El sistema operativo, como se ve, es uno basado en Unix. Se evita la utilización de un sistema Windows por la versatilidad que ofrece un sistema Linux en cuanto a configuraciones y por la mayor gama de tecnologías y lenguajes que pueden utilizarse en éste.

La plataforma sobre la que se ha desarrollado (**Ubuntu 8.04**), así como

la tecnología de middleware utilizada (**Ice 3.3.1**) son requisitos iniciales de la aplicación, a los que hay que unir uno más importante, y es el que limita la comunicación de **IceStorm** al lenguaje **C++**. El resto de aplicación (captura de la cámara, volcado a disco, interfaces gráficas), están realizados en **Java**, por tanto, unir estas dos partes supone un problema en sí mismo, que se comentará en las secciones siguientes. Como es evidente, separar la implementación de la aplicación en dos lenguajes distintos, aumenta significativamente la complejidad de la misma, a la vez que reduce la eficiencia, la velocidad y el rendimiento de la aplicación. El objetivo es probar **Ice** tanto en un ámbito **Java** como en un ámbito **C++,** además de poner a prueba los "*mappings*" (enlaces) correspondientes que realiza **Slice** (el lenguaje que proporciona independencia de plataforma) sobre **Java** y sobre **C++.** A continuación se muestra un diagrama en el que se representan las distintas partes de la aplicación:

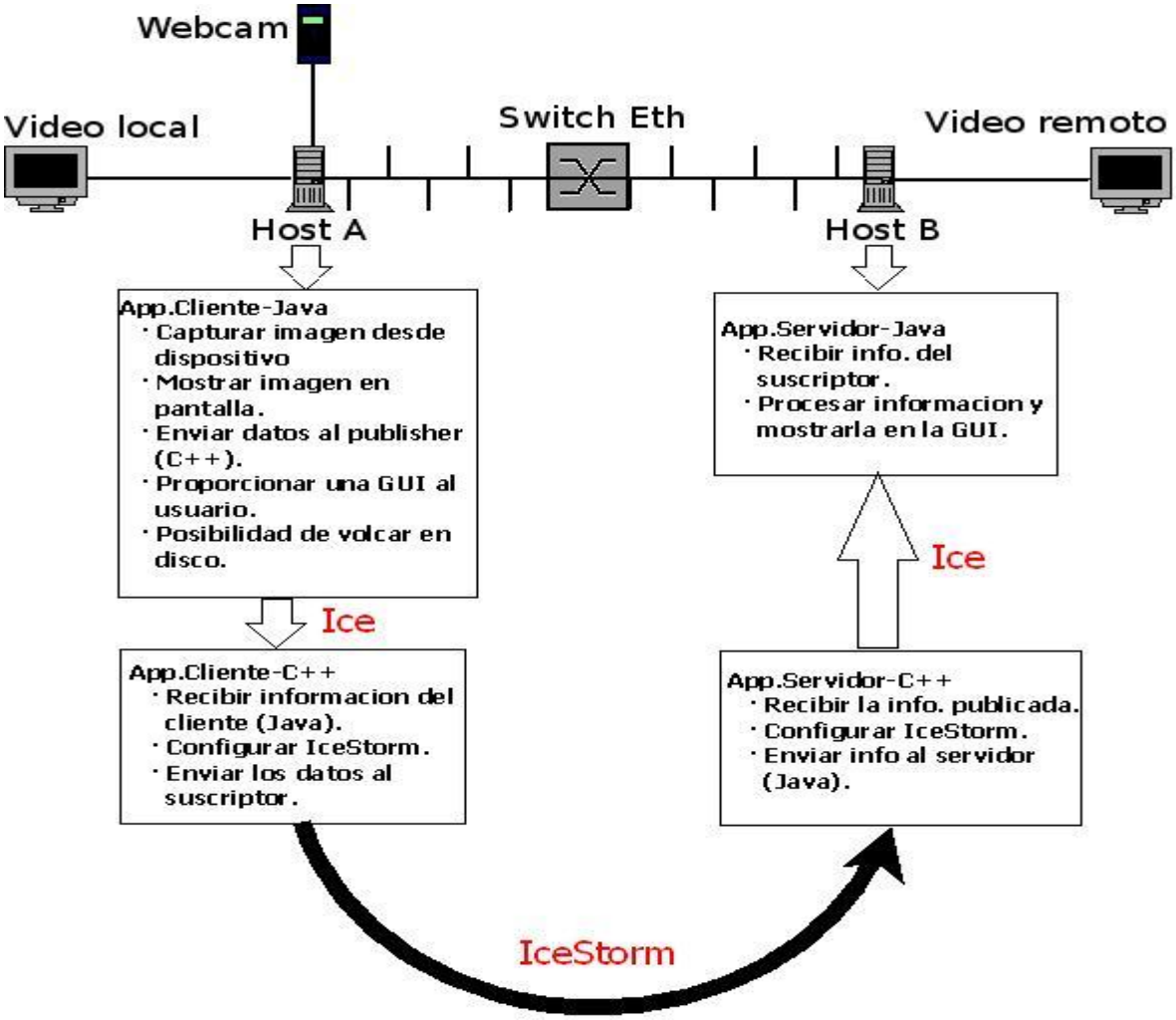


Figura 18: Diagrama de distribución de funcionalidades.

Una vez detallado el entorno en el cual se ha desarrollado, probado y desplegado la aplicación de videovigiliancia remota, y los requisitos más fuertes que se han establecido y que conducen gran cantidad de las decisiones a tomar en la solución del problema, se pasa a presentar cada una de las tareas que han conformado la aplicación, describiendo el proceso que se ha seguido para solucionar cada uno de los problemas encontrados. Se detallan las decisiones tomadas, el diseño realizado y el código relevante correspondiente a la implementación de cada parte.

3.2.1 Problema de la captura y solución

Se desea diseñar una aplicación que sea capaz de acceder a un dispositivo de entrada para tomar información de éste, que más tarde se tratará de diversas formas. Se plantean diversos problemas, el primero de ellos es, obviamente, ver qué nos ofrece el dispositivo del que se dispone y estudiar la viabilidad de utilizar una tecnología u otra para acceder a él.

Se dispone de una **Webcam Creative Instant Camera**, modelo **VF-0040**, la cual, como es de esperar contiene drivers para plataformas Windows pero nada para sistemas operativos derivados de Unix. Aquí, por tanto, aparece el primer problema. Para el acceso y control de dispositivos de entrada-salida como es el caso de la webcam, existen, entre muchas otras, las siguientes posibilidades:

- Windows:
  - **.NET** -> **DirectShow**
- Unix:
  - Basadas en **Java**:
    - **JMF**(Java Media Framework).
    - **FMJ**(Freedom for Media in Java).
  - Basadas en librerías de Unix:
    - **GStreamer**.
    - **Video4Linux**.
  - MIX:
    - **Video4Linux4Java**.

Puesto que desde el inicio se ha dicho que la plataforma sobre la que se desarrollara será un sistema basado en Unix, se eliminan las posibilidades referentes a sistemas Windows. Buscando la mayor facilidad en el acceso al dispositivo, se elige la opción de utilizar el driver **Video4Linux (V4L). V4L** es un driver genérico que se encuentra en la mayoría, sino en todas, las distribuciones

Linux cuyo kernel sea superior a la **versión 2.6** (ver "`man` **`V4L`**"), y que soporta la cámara que se tiene para las pruebas.

Este driver se dice genérico en tanto que abarca la posibilidad de utilizar un amplio número de dispositivos de vídeo, no es un driver concreto para un modelo de cámara, sino un medio para acceder a muchos tipos de dispositivos de naturalezas bien distintas. Para el caso de la aplicación, la webcam de la que se dispone es compatible con el driver **Video4Linux**, por tanto el sistema reconoce el dispositivo una vez conectado, montándolo en el directorio `/dev`.

Tras explorar las posibilidades que ofrecen los sistemas Linux más modernos, el problema de acceso al dispositivo se desvanece con la utilización de Video4Linux.

Aplicando los requisitos de la aplicación, se acotan en gran medida el número de opciones restantes. La solución elegida finalmente de entre las restantes, es la utilización del **API V4L4J**. Este API provee de mecanismos para acceder a dispositivos basados en el driver **V4L**, a la información específica de éstos, controlar ciertos parámetros de la imagen, y tomar información del dispositivo (todo mediante los drivers nativos de **V4L**, existentes en el sistema operativo). Estos drivers nativos están escritos en **C**, como se indicó en la tabla de arriba, pero el API **V4L4J** provee precisamente de mecanismos para utilizar éstos con **Java**, mediante el empleo de **JNI** (Java Native Interface) para hacer un "*wrapper*" (envoltorio) de estos drivers y ofrecer la misma funcionalidad desde el lenguaje **Java**. Con la utilización de esta librería se cumple con otro requisito de la aplicación, que es la utilización de la tecnología **Java** en la lógica de acceso y control del dispositivo.

V4L4J es un API relativamente sencillo que ofrece unas posibilidades bastante limitadas (acceso, control básico del dispositivo y toma de información), pero que para el objetivo del proyecto son más que suficientes. Sería un error utilizar API's más complejos como podrían ser **Gstreamer**, por la dificultad que habría supuesto llegar al mismo punto que con **V4L4J** y porque las funcionalidades estrella de estos frameworks nunca habrían sido necesarias y por tanto no se habrían utilizado en ningún caso. Además, **V4L** dota de una relativa sencillez a la aplicación, evitando el uso de plataformas más amplias como las comentadas anteriormente, y reduciendo esto a la utilización de un par de librerías.

Una vez justificada la decisión de utilizar **V4L4J** para el acceso y control de la cámara se van a describir aquellas partes del código que son relevantes para entender el funcionamiento de la librería.

Lo primero que debe hacerse es configurar los parámetros con los cuales se desea acceder y manipular la información de la cámara. Para esto se programa un bean (**`WcamBean`**) en el cual se encuentran todas las propiedades configurables de la cámara en forma de atributos.

Como puede observarse se encuentran unos valores para dichos atributos, estos son los valores por defecto. No obstante el valor de todos y cada uno de los atributos que aparecen, es modificable a través de la interfaz gráfica que se ofrece al usuario cuando se ejecuta la aplicación.

Por supuesto, **`WcamBean`** también incluye los métodos **`get/set`** para acceder a los atributos. Los atributos son:

```
private String device = "/dev/video0";
private int width = 640;
private int heigth = 480;
private int standard =
au.edu.jcu.V4L4J.V4L4JConstants.STANDARD_WEBCAM;
private int channel = 0;
private int quality = 60;
```

Los valores de los atributos anteriores son los usados por defecto para acceder a la webcam, y una vez configurada ésta con dichos parámetros, se procede del siguiente modo:

1-. Importación de las clases necesarias para tratar con el dispositivo:

```
//Clases a importar.
//Pertenecientes al API estándar de Java.
```

```
import java.nio.ByteBuffer;

//Pertenecientes al API concreto de la librería V4L4J.

import au.edu.jcu.V4L4J.FrameGrabber;
import au.edu.jcu.V4L4J.VideoDevice;
import au.edu.jcu.V4L4J.InputInfo;
import au.edu.jcu.V4L4J.exceptions.V4L4JException;
```

2-. Se indica la ruta del dispositivo y se crea un objeto **`VideoDevice`** que implementa los métodos para manejar el dispositivo al más alto nivel.

```
/**
El parámetro del constructor VideoDevice es el valor del atributo Device, que por defecto será /dev/video0 (accedido desde WebcamBean [wcb]), el lugar en el que V4L monta los dispositivos de entrada.
*/

VideoDevice vd = new VideoDevice(wcb.getDevice());
```

3-. El siguiente paso es obtener un objeto **`FrameGrabber`**, que es el que permitirá acceder a la información del dispositivo. Para este fin es necesario decidir en qué formato se desea obtener la información, puesto que esto determinará el método que debe utilizarse para obtener el objeto **`FrameGrabber`**. En este caso, se ha decidido tomar la información comprimida en **JPEG**, debido a que no es un requisito importante la calidad de imagen y en cambio sí es importante manejar la menos información posible, en tanto que esta información va a viajar por la red y se quiere conseguir los menores valores de retardo posibles. Esta decisión planteaba un posible problema, y era el hecho de que si la información obtenida de la cámara se encontraba comprimida en **JPEG**, los sucesivos *frames* que fueran accediéndose no iban a tener una longitud igual necesariamente, de hecho, lo más probable es que fueran distintos en cada caso, según la compresión que se aplicase. En las siguientes secciones, donde se trata el problema de los *mappings* entre **Java** y **C++** se irá esclareciendo por qué finalmente este posible problema no es tal, y por tanto lo más conveniente es una compresión en **JPEG**.

```
/**
Se aplica sobre el objeto VideoDevice obtenido anteriormente el método
```

getJPEGFrameGrabber(), el cual devuelve un objeto FrameGrabber, con el que puede operarse sobre la cámara. Como se observa, es necesario incluir todos aquellos parámetros de configuración con los cuales queremos acceder a la información de la cámara, y estos se toman del bean mostrado anteriormente.
*/

```
FrameGrabber fg = vd.getJPEGFrameGrabber(wcb.getWidth(),
wcb.getHeigth(), wcb.getStandard(), wcb.getChannel(),
wcb.getQuality());
```

4-. Una vez conseguido el objeto **`FrameGrabber`** asociado al dispositivo en cuestión y con los parámetros de configuración necesarios, no se tiene más que jugar con el ciclo de vida de este objeto, que es tan simple como indicar cuando comienza y termina la captura. Y con la captura comenzada, puede accederse finalmente a la información que está capturando el dispositivo con un método específico de **`FrameGrabber`**, que devolverá una imagen comprimida en **JPEG**, debido a lo explicado anteriormente.

```
/**
El método getFrame(), aplicado al objeto FrameGrabber nos devuelve un ByteBuffer
que tiene la información perteneciente al frame que la cámara ha capturado en el
momento de la llamada. Nos interesa tener un array de bytes, puesto que es una
estructura mucho más manejable en Java, lo cual nos permitirá operar mejor con
ella a la hora de pintar en disco.
*/

ByteBuffer bb;
byte[] b;
try{

        //Se inicia el ciclo de vida del objeto FrameGrabber con una llamada
        como la siguiente, siendo fg, el objeto FrameGrabber.

        fg.startCapture();

        //boolean goon es una variable que controla cuando hay que capturar
        //información y cuando no.

        while(goon){
```

```
            //Guardamos en ByteBuffer bb la información que capturamos en
            //el momento de la llamada.

            bb = fg.getFrame();

            //ByteBuffer.limit() nos da el límite del buffer, con lo que obtenemos su
            //dimensión, y con esto creamos el array de bytes con la dimensión
            //adecuada en cada captura.

            b = new byte[bb.limit()];

            //Finalmente volcamos la información del ByteBuffer en el array de bytes
            //que hemos creado.

            bb.get(b);

            //dataUnit es un nuevo bean, del que no comentaremos mucho mas por
            //el momento, solo es interesante saber que es el lugar donde
            //depositaremos la información que vamos obteniendo en la captura.

            dataUnit.setData(b);
        }
}

//Se trata la posible situación en la que haya una excepción, ya sea por no encontrar
//el dispositivo, o por intentar crear el objeto FrameGrabber con parámetros no
//válidos.

catch(V4L4JException e) {
            e.printStackTrace();
                System.out.println("Failed to capture image");
}

//En el bloque finally se incluyen aquellas operaciones que deben realizarse siempre
//que termine de capturarse, estas son stopCapture(), que cierra correctamente el
//ciclo de vida del objeto FrameGrabber. Las operaciones releaseFrameGrabber() y
//release() se ocupan de desligar el objeto FrameGrabber del objeto VideoDevice, y
//el objeto VideoDevice de la cámara respectivamente. Esto se realiza para liberar
//recursos.
```

```
finally{
          fg.stopCapture();
          vd.releaseFrameGrabber();
          vd.release();
}
```

5-. Al final de esta serie de pasos, se ha conseguido la información correspondiente a un *frame* capturado por la cámara. Se ha transformado esta información en una estructura mucho más manejable por la tecnología **Java** como es un *array*, y se ha depositado este *array* en un bean, al cual podrán acceder las partes de la aplicación que lo necesiten.

Como se observa, la utilización del API **V4L4J** es realmente sencillo, sin embargo, es cierto que es delicado decidir si es conveniente o no información de tamaño constante, y el tipo de estructura en el que se encuentre esta información. En los siguientes capítulos se comentará por qué la decisión más óptima es en este caso obtener la información comprimida en **JPEG** y representada en un *array* de **Java**, pero básicamente se observará que hará falta pasar datos de una aplicación **Java** a una **C++** mediante **Slice** de **Ice**, y para esto la estructura más apta de **Java** es un *array* de bytes. Además, se comprobará cómo la variabilidad de la longitud de los datos termina por ser irrelevante, y por esto se prefiere finalmente compresión **JPEG**.

3.2.2 Diseño multihilo

Como es de esperar, en la aplicación que se propone, se encontrará un problema relacionado con la independencia de las tareas que la aplicación está destinada a realizar, lo cual lleva a pensar rápidamente en programación multihilo, y los problemas que subyacen a esto.

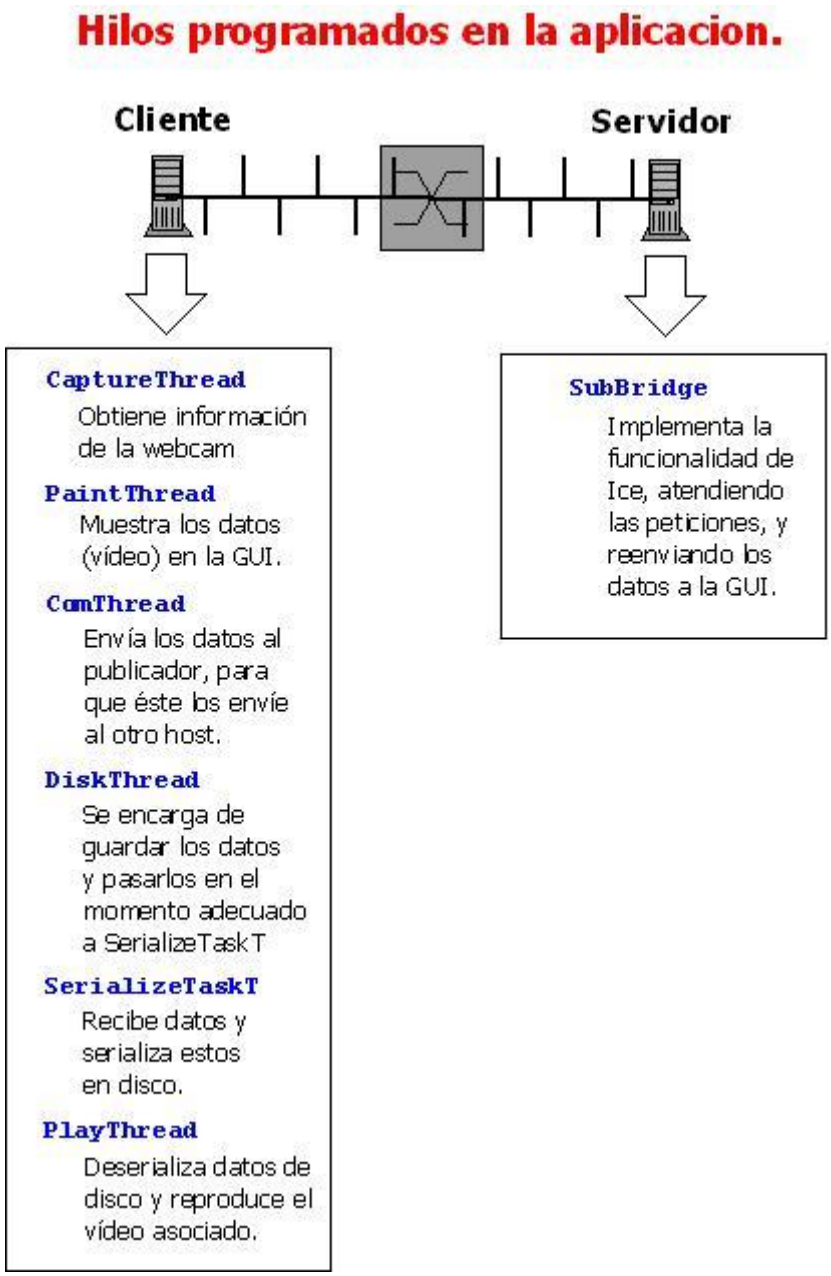


Figura 19: Hilos programados de la aplicación.

En la aplicación, tras analizar detenidamente las tareas que ésta debe realizar, se necesitan los hilos que aparecen en la figura anterior. La justificación de su uso se ofrece a continuación, para pasar después a la implementación concreta.

Parte Cliente:

- La funcionalidad de la aplicación es accesible al usuario mediante el empleo de una interfaz gráfica desarrollada con **SWING**, un API específico de la tecnología **Java** cuyo fin es proveer de un mecanismo potente para la realización de interfaces de ventanas de usuario. Esta interfaz responde a las interacciones del usuario mediante la generación de eventos, que son tratados por ciertos métodos, cuya implementación es tarea del programador. En este punto se encuentra el primer hilo de la aplicación, puesto que los eventos generados por el usuario son despachados por un hilo interno a la tecnología **SWING** (más bien, hereda el modelo de eventos de **AWT**). Este hilo no es configurable por parte del programador y es conocido por **`Event Dispatch Thread`**.

- Por otro lado, se tiene que capturar información de una cámara, de la manera que se ha descrito anteriormente. Esta información extraída, será la fuente de casi el resto de hilos que van a comentarse. En tanto que se pretenda ver el vídeo en tiempo real, no se puede dejar de capturar información, por lo que aquí entra el primer hilo diseñado específicamente para la aplicación. La función de este primer hilo es la de cumplir con la tarea de acceso, configuración, y captura de información a través de la cámara. A este hilo se le llamará **`CaptureThread`**.

- La información generada por el hilo **`CaptureThread`** debe ser representada localmente en la interfaz de usuario **SWING**. Para esto se dispone de otro hilo, cuyo nombre es "**`PaintThread`**", el cual, como su propio nombre indica se ocupa de acceder a la información que está generando **`CaptureThread`**, realizar las conversiones necesarias para disponer de una información representable en un componente **SWING** y plasmar finalmente esta información en forma de sucesión de imágenes.

- En el momento en el que en la máquina local se comienza a observar la captura, ésta misma debe ser visible con un retardo mínimo en el nodo destino. Esta tarea es con diferencia la de mayor complejidad del proyecto, sin embargo no es el acceso a la información una tarea compleja en este caso. Simplemente se dispone de un hilo llamado **`ComThread`** que recogerá la información que el hilo

`CaptureThread` está generando. Además de esto, este hilo se encarga de establecer una comunicación con un "*bridge*" (puente software) implementado en el lenguaje **C++,** que es el encargado de ofrecer la funcionalidad del "publisher".

- La última tarea que proponía la aplicación era la de proveer un sistema para guardar la información que se está capturando en un dispositivo con memoria no volátil. Se decidió guardar en el disco duro por ser el dispositivo de almacenamiento persistente más rápidamente accesible desde el sistema operativo. Esto supone un problema en sí mismo, puesto que se trata con operaciones de **I/O**, las cuales son notablemente más lentas que el resto de operaciones que trata la aplicación. (Más incluso que comunicación remota). La solución al problema fue establecer un hilo, **`DiskThread`**, que se encargase de guardar la información en un búfer, modelado como un **`Vector`** de **Java**, el cuál crece hasta que posee una cantidad de información dada. En este momento este hilo pasa la información a otro hilo **`SerializeTaskT`** el cual se encarga de serializar en paralelo la información que ha recibido, mientras el hilo **`DiskThread`** sigue acumulando los datos de la captura. La cantidad de información con la que **`DiskThread`** llama a **`SerializeTaskT`** se calcula de forma empírica de modo que sea posible para **`SerializeTaskT`** serializar dicha información antes de que **`DiskThread`** le envié más. Cuando se da por terminada la captura, se ofrece al usuario un cuadro donde puede poner nombre al vídeo y guardarlo pertinentemente en el disco. La siguiente ilustración aclara la idea.

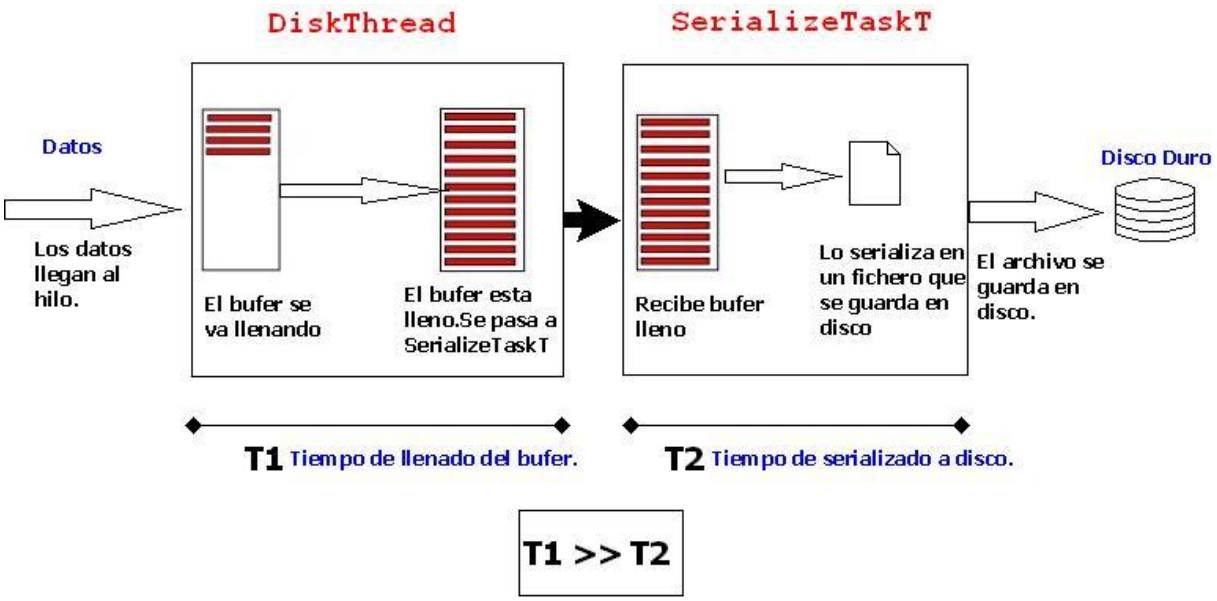


Figura 20: Funcionamiento de DiskThread y SerializeTaskT.

- El hilo comentado en el punto anterior, **`SerializeTaskT`**, es un hilo que implementa una funcionalidad muy sencilla. Simplemente recibe información en forma de vector, y cuando es invocado, serializa esta en disco, asociando al fichero destino un **UUID** (Universally Unique Identifier), de modo que no pueda solapar con cualquier otro fichero existente. El nombre del fichero final se guarda en memoria, y se procede del mismo modo con todos los ficheros que sean necesarios guardar, pertenecientes a la misma captura. Cuando el usuario decide acabar con la captura, lo que ocurre es que un objeto que mantiene en memoria la referencia a todos los sub-archivos del vídeo capturado, es serializado y guardado asimismo como fichero en disco, de modo que sea recuperable más tarde.

- Del resultado del hilo anterior se intuye que hace falta incluir funcionalidades de deserialización, que sean capaces de restablecer un objeto **`Vector`** de **Java** a partir de ficheros generados por nuestra propia aplicación, de modo que pueda reproducirse un vídeo grabado anteriormente. Para esta tarea se provee de otro hilo llamado **`PlayThread`**. Antes de comentar las funciones que realiza, se puede reflexionar sobre si realmente es necesaria o no la inclusión de este hilo. En la interfaz de usuario se dispone únicamente de un espacio para el vídeo, de modo que puede estar mostrándose la captura actual, o la reproducción del archivo seleccionado. Pues bien, se tomó como consideración que al reproducir un vídeo, se mostraría éste en la interfaz, pero que seguiría siendo necesario mostrar la captura actual en el nodo remoto. Debido a esta consideración se decidió incluir la funcionalidad de reproducción en un nuevo hilo. El hilo por tanto, se encarga de deserializar el fichero seleccionado por parte del usuario (el que mantiene las referencias a los distintos sub-archivos), acceder a cada sub-archivo, juntar la información resultante en un objeto **`Vector`** de **Java** y representar la información capturada en un componente **SWING**.

Parte Servidor:

- En la parte servidora, como se verá más adelante, se encuentra un módulo encargado de recibir la información que envía la parte cliente. Este módulo se bloquea, y cada vez que llegan datos se llama a una función que se encarga de gestionar dicha información. Debido a que se establece también una interfaz de usuario en esta parte, para mostrar el vídeo, es necesario que no se bloquee la ejecución de la aplicación. Este hecho lleva a incluir la funcionalidad del módulo que recibe la información, en un hilo, cuyo nombre es **`SubBridge`**.

Vistas las necesidades de la aplicación, a continuación se pasa a analizar las partes claves de cada uno de los hilos, así como a describir cómo se han solucionado algunos problemas que han surgido a la hora de la implementación de la aplicación multihilo, y los mecanismos que se han seguido para dar cohesión a ésta.

Antes de pasar a la implementación concreta de cada hilo, se muestra en la siguiente ilustración los hilos con sus respectivos métodos y atributos.

Figura 21: Hilos de la aplicación, atributos y métodos.

Hilo 1: "Event Dispatch Thread".

El hilo encargado de atender los eventos generados en la interfaz de usuario **SWING** no es configurable por parte del usuario, lo cual no exime la responsabilidad que tiene el programador de atender a este hilo.

Se presupone que es deseable proveer al usuario de una interfaz con una rápida respuesta, de modo que sea más amigable la interacción con la interfaz que se ofrece. No son deseables las esperas que se producen a veces cuando se pulsa un botón en una aplicación y la sensación que se percibe es que la aplicación se ha bloqueado. Estos parones son en bastantes ocasiones producidos debido a que el hilo que se encarga de gestionar la lógica asociada a dicho control se queda bloqueado ejecutando las órdenes necesarias, de modo que al usuario le queda la sensación de aplicación ralentizada y lenta. Por tanto, para nuestra aplicación se ha tenido en todo momento en cuenta este hecho, derivando, en el caso de necesidad de un procesamiento pesado, la lógica asociada a los botones que se presentan en hilos distintos, de modo que vuelva rápidamente el control al hilo de atención a eventos, y la lógica asociada al control se ejecute "paralelamente" en otro hilo.

Un ejemplo de lo comentado son las acciones realizadas tras pulsar el botón START de la interfaz de usuario. Cuando se pulsa START, la aplicación debe realizar las siguientes acciones: iniciar la recuperación de los parámetros de configuración del dispositivo de entrada (webcam), acceder al dispositivo en concreto, configurar éste y extraer la información. A su vez, debe estar recuperando esta información para mostrarla en pantalla y para enviarla al ordenador remoto. Todas estas tareas obviamente suponen un tiempo "apreciable" en la aplicación. Si se pretendiesen realizar todas ellas dentro del método **`actionPerformed(ActionEvent e)`**, la aplicación se quedaría bloqueada hasta que terminasen todas, lo cual es un error muy grave. Por tanto, se delega la responsabilidad de realizar estas tareas a otros hilos, que son los descritos más arriba.

```
//El método actionPerformed(ActionEvent e) es llamado por el hilo de atención a
//eventos cuando se produce uno, quedándose la ejecución del hilo de eventos
//bloqueada hasta que este método finalice.
```

```
public void actionPerformed(ActionEvent e){

//El método getActionCommand(), comprueba que control de la interfaz ha sido
//accionado.

if(e.getActionCommand().equals("START")){

        //Se toma la información de configuración de la cámara. Para ello se
        //aplica el método getWebcamBean(), sobre el objeto WebcamView
        //(wcv), el cual posee como atributo precisamente dicho bean.

        WcamBean wcb = wcv.getWebcamBean();

        //Se crean los hilos necesarios que implementaran la lógica necesaria.

        capT = new CaptureThread(wcb, dataUnit, true);
        pT = new PaintThread(vid, dataUnit, true);
        comT = new ComThread(dataUnit, true);

        //Se arrancan dichos hilos

        capT.start();
        pT.start();
        comT.start();
        }
...
}

//Al finalizar el método, el hilo de atención a eventos vuelve a estar ocioso para
//poder atender a otros eventos que pueda generar el usuario.
```

HILO 2: "CaptureThread".

La funcionalidad que tiene que realizar este hilo, así como el modo en que la implementa están descritos en la sección anterior (3.2.1), no obstante, es necesario indicar en este punto algunas consideraciones más, como son el hecho de que para la aplicación éste debe ser el primer hilo que entre en funcionamiento, puesto que la mayoría del resto dependen activamente de la información que éste

recupere de la cámara.

Es necesario indicar también en este momento un punto de extrema importancia para la aplicación, como es el acceso a la información. Se tiene a **`CaptureThread`** que escribe información en un bean, llamado **`FrameBean`**, y después se encuentran el resto de hilos que deben acceder a esta información, **`PaintThread`**, **`DiskThread`**, y **`ComThread`**. El acceso a la información puede ser un problema si mientras **`CaptureThread`** está escribiendo la información en el bean, cambia la ejecución a alguno de los otros hilos y pretenden precisamente leerla, puesto que se producirían errores de consistencia de memoria, lo cual es un grave problema. Sin embargo, la solución es bien sencilla, la más directa es hacer que el método encargado de escribir la información en el bean sea sincronizado, de modo que se asegure el cerrojo del objeto durante la escritura y ningún otro hilo de la aplicación pueda acceder a este atributo.

Se incluye el código relevante del bean:

```
[...]
//Bean.

public class FrameBean{

//Atributo que modela el array de bytes extraído de la cámara.

private byte[] data;
[...]

//Método para acceder (lectura) a la información.

public byte[] getData(){
          return data;
}

//Método sincronizado para asegurar el acceso (escritura) de un único /
/hilo de ejecución a la información.

public synchronized void setData(byte[] data){
          this.data = data;
```

```
        }
        [...]
```

HILO 3: "PaintThread".

Este hilo, como se ha comentado anteriormente es el encargado de mostrar las imágenes a través de un componente **SWING** que se encuentra en la interfaz de usuario. Tiene que ser capaz de ofrecer esas imágenes a una velocidad tal que para el ojo humano sea apreciable un flujo continuo, es decir, un vídeo. La televisión de alta calidad da de 25 a 30 frames/segundo. En la demo se pretende obtener una frecuencia de imágenes que resulte admisible para el usuario. Para esto, el objeto **SWING** en el cual se representan las imágenes es un **`Jlabel`**, y los pasos que se siguen son simplemente convertir el *array* de bytes, que es la información que obtiene "**`CaptureThread`**" en un objeto **`ImageIcon`** de **Java**. En este caso el API nos ofrece directamente un método que recibe por argumento un *array* de bytes, y él es encargado de establecer la imagen que sea necesaria. Una vez más, se observa la comodidad de mantener la información obtenida de la cámara guardada en un *array* de bytes, que es una estructura estándar del lenguaje **Java**, en favor de hacerlo en un objeto **`ByteBuffer`**, ya que en este último caso no nos quedaría otro remedio que transformarlo a un *array* de **Java** finalmente, introduciendo un pequeño retardo adicional.

A continuación se muestra el código relevante:

```
//El método run del hilo PaintThread

public void run(){
        byte[] data = null;

        //Como en el caso anterior, la variable goon controla la ejecución del hilo.

        while(goon){

        //Esta comprobación se realiza para asegurar que la muestra de datos
        //que se toma sea distinta a la tomada anteriormente. Podría darse una
        //situación en la que PaintThread accediese un numero de
        //veces a los datos, sin que CaptureThread hubiera refrescado estos, y
```

```
            //en este caso se estaría perdiendo tiempo de CPU.
                        if(data == dataUnit.getData()) continue;

                        //Se toman los datos a representar.

                        data = dataUnit.getData();

                        //Al inicio, mientras la cámara esta configurándose, la
                        //información tiene un valor NULL, aun no existe,
                        //por ello es necesario comprobar que esto no sucede.

                        if(data != null){

                        //viewLabel es el objeto Jlabel perteneciente a la interfaz de
                        //usuario. El método setIcon()
                        //establece un objeto ImageIcon como imagen. Y el
                        //constructor ImageIcon(byte[]) ofrece un objeto
                        //ImageIcon a partir de un array de bytes.

                                    viewLabel.setIcon(new ImageIcon(data));
                        }

            }

//Finalmente, una vez se sale de la ejecución, es decir, cuando no es necesario
//pintar mas, se pinta un objeto ImageIcon vacío. Esto borra la última imagen que
//existiera en el Jlabel, dejando esta vacía.

            viewLabel.setIcon(new ImageIcon());

}
```

HILO 4: "ComThread".

`ComThread` es el hilo encargado de la realización de la primera parte de la comunicación. El funcionamiento es similar a los anteriores, este hilo debe

acceder a la información, lo cual consigue exactamente del mismo modo que el resto de hilos. Tras esto, utiliza una llamada a método remoto en la cual incluye los datos pertenecientes a la imagen. Toda la problemática de la comunicación se tratará más adelante, por ahora sólo interesa ver cómo realiza esta tarea, para lo cual se incluye el código.

```
//El método run del hilo en cuestión.

public void run(){
          byte[] data = null;

          //Como en el resto de hilos comentados, la variable goon controla la
          //ejecución de este.

          while(goon){
                    if(data == dataUnit.getData()) continue;

                    //Accedemos en lectura a los datos.

                    data = dataUnit.getData();

                    //Comprobamos que no tengamos unos datos vacíos.

                    if(data != null){
                              //imageData es el objeto remoto, proxy en la
                              //jerga Ice. Report() es el método remoto, interfaz
                              //remota en la jerga Ice, y aquí realizamos la
                              //llamada en la cual pasamos los datos

                              imageData.report(data);
                    }
                    [...]
```

Tras el código comentado se incluye una sentencia **`sleep`** en el código. Esta decisión viene tomada, con la intención de buscar una rapidez mayor en la aplicación. La sentencia mantiene el hilo sin ejecución durante ¼ de décima de segundo. La justificación de esta decisión viene dada por la naturaleza de la aplicación que se está desarrollando. Desde luego, está claro que hay unos hilos que tienen mayor importancia para la aplicación que otros, como es el caso de

`PaintThread`, por ejemplo, o como es el hilo de atención a eventos de **SWING**. Por tanto intuitivamente es necesario pensar que estos hilos tendrán unas prioridades mayores que el resto. El problema de esto es que la prioridad del hilo de atención a eventos es algo que no puede configurarse. Por esto se recurre a la sentencia **`sleep`**, debido a que indirectamente haciendo descansar a unos hilos más que a otros, se favorece la ejecución de los hilos que no descansan, aumentando por tanto su "prioridad". En muchos casos, no es posible realizar esto, pero en este caso concreto debido a la naturaleza de la aplicación, sí es posible, puesto que una pausa de un cuarto de décima de segundo en un flujo de vídeo es prácticamente inapreciable por el ojo humano.

```
try{

//Hacemos que el hilo descanse 25 milésimas → 0,025 segundos.

          sleep(25);
}
catch(java.lang.InterruptedException e){
          System.out.println("Excepcion en sleep: "+e.getMessage());
}
```

Como se ha dicho, la parte de comunicación se detallará más adelante, pero es importante indicar ahora que la llamada al método remoto **`report()`** no se bloquea hasta esperar una respuesta puesto que se configura una comunicación unidireccional. Por tanto, una vez realizada la llamada vuelve el flujo de ejecución a la aplicación, evitando que se quede bloqueada hasta esperar una respuesta (un **ACK** o información de retorno).

HILO 5: "DiskThread".

El funcionamiento de este hilo ha sido explicado anteriormente. Básicamente se encarga de guardar la información en un **`Vector`** en memoria, el cual será accedido por otro hilo que se encargará de ir serializándolo a disco. El acceso a la información en lectura se realiza exactamente del mismo modo que en

el resto de hilos vistos. Nuevamente, se introduce una sentencia **sleep**, que para la ejecución del hilo durante 50 milésimas, esto se justifica del mismo modo que para el hilo **ComThread**. Además puede verse como un sistema muy rudo de muestreo, ya que se están tomando muestras como mínimo cada 50 milésimas. Si se mantiene esta tasa a la hora de reproducir el vídeo, es decir, entre muestra y muestra se esperan 50 milésimas de segundo, se verá una reproducción a una velocidad muy similar a la de la captura. Intuitivamente se aprecia que no será a la misma velocidad, puesto que en la captura a las 50 milésimas de espera mínima, habrá que sumarle el tiempo de ejecución de los hilos que tomen la ejecución entre medias, y este tiempo no va a ser constante, no obstante, el defecto del ojo humano vuelve a hacer viable esta situación, permitiéndose realizar estas esperas sin sacrificar apenas la calidad del vídeo.

```
public void run(){

        //mem es el Vector Java donde se guardaran las muestras obtenidas. Se
        //inicializa a 3000, que es un valor  suficientemente grande para no
        //sacrificar a menudo tiempo para redimensionar el Vector, además la
        //tasa de crecimiento es de 500, cifra que busca la misma finalidad que
        //la anterior.

        mem = new Vector(3000, 500);
        byte[] data;
        while(goon){

                //Acceso a datos en lectura

                data = dataUnit.getData();

                //Guardamos los datos en la memoria temporal.

                mem.addElement(data);

                //La espera de 50 milésimas comentada anteriormente.

                try{
                        sleep(50);
                }
                catch(java.lang.InterruptedException e){
                System.out.println("Excepcion en sleep:
```

```
"+e.getMessage());
                    }
          }
```

```
//Antes de terminar la ejecución, se hace un fix al vector para liberar aquel espacio
//reservado que no ha utilizado.

mem.trimToSize();
```

HILO 6: "SerializeTaskT".

El hilo **`SerializeTaskT`**, implementa la funcionalidad requerida para serializar la información que reciba a disco, y llevar un control de los distintos subarchivos que ha ido serializando en un nuevo fichero, que será desde el cual pueda accederse a dicha información y reproducir el video concreto. A continuación el código.

```
//Primeramente se muestran los atributos que se utilizan en el hilo. El Vector mem,
//guarda la información temporal que se va volcando, en memoria. El Vector files,
//guarda el "path" a cada uno de los subarchivos que se serializan. El String
//basepath supone el lugar de origen donde se guardan y leen los ficheros. IOData
//io, es la clase encargada de serializar y deserializar ficheros. La variable id es un
//identificador que se añade al nombre de los subficheros para reconocer el orden
//en el que hay que deserializarlos. El boolean goon controla la ejecución del hilo.

private Vector mem;
private Vector files = new Vector(10,1);
private String basepath = "videoHome";
private IOData io = new IOData();
private int id = 10000;
private boolean goon = true;
private boolean ctr = false;

public void setGoon(boolean goon){
          this.goon = goon;
}
```

```
public void setCtr(boolean ctr){
          this.ctr = ctr;
}

public void setMem(Vector mem){
          this.mem = mem;
}

public Vector getFiles(){
          return files;
}

public SerializeTaskT(){
}

//El método run del hilo, que es llamado por DiskThread cada vez que tiene
//información suficiente para serializarse.

public void run(){

          //Se genera un UUID para asegurar que no haya dos ficheros con mismo
          //nombre.

          String aux = "/."+(UUID.randomUUID()).toString();

          //Se junta el path base, que es “videohome” con el UUID generado y el id
          //que va incrementándose en tanto se serializen mas ficheros a disco.

          String path = basepath+aux+id;
          System.out.println(path);

          //Si la operacion de serializado tiene exito, entonces se guarda el
          //nombre del subfichero en el Vector files, que sera accedido más tarde
          //por PlayThread para deserializar dichos ficheros y reproducir el video.

          if(io.serialize(path,mem)) files.addElement(path);
          id++;
          }
}
```

HILO 7: "PlayThread".

El hilo encargado de reproducir los vídeos que anteriormente hayan sido guardados a disco. Lo primero que hace es ayudarse de un objeto de la clase **`IOData`** para deserializar el fichero. Antes de ver el funcionamiento concreto, que es muy sencillo, se muestra el código relevante de la clase **`IOData`**.

```
[...]

//Recibe por parámetro el path donde tiene que guardar el fichero resultado de la
//operación, y el vector que contiene los datos.

public boolean serialize(String path, Vector mem){
[...]
          //Crea el objeto de salida donde se guardará la información.

          ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream(path));

          //Escribe en disco el objeto.

          out.writeObject(mem);

          //Libera buffers y cierra el stream.

          out.flush();
          out.close();
[...]

//Este método devuelve un Vector, el cual es resultado de deserializar el fichero que
//se encuentra en la ruta definida por path

[...]
public Vector deserialize(String path){
[...]

          //Abre un stream de entrada.
```

```
            ObjectInputStream in = new ObjectInputStream(new
FileInputStream(path));

            //Lee el objeto escrito en disco y lo guarda en mem, que es un Vector

            mem = (Vector)in.readObject();

            //Cierra flujos.

            in.close();
```

De este modo se ofrece una forma sencilla de serializar y deserializar la información que se tiene en el **`Vector`**. Y de este mecanismo se vale el hilo **`PlayThread`**, para recuperar la información y reproducirla en la interfaz **SWING**, como se ve a continuación.

```
//Método run del hilo en cuestión.

public void run(){

            //Recuperamos el vector con la información relativa al vídeo que se va a
            //reproducir a partir de path, que es un String que indica la
            //ruta al archivo en disco que contiene dicha información.

            Vector mem = iodata.deserialize(path);

            //Recorremos el Vector por completo...

            for(int i = 0; i<mem.size(); i++){

            //...accediendo al array en cada posición y pintando la imagen en
            //SWING.

            viewLabel.setIcon(new ImageIcon((byte[])mem.elementAt(i)));

                        //Tras cada iteración, realizamos esperas de 50 milésimas,
                        //que son exactamente las mismas que esperamos a la hora
                        //de capturar la información. De este modo aseguramos
                        //evitar situaciones en las que el vídeo reproducido se vea
```

```
                    //más lento o mas rápido que cuando se realizo la captura.

                    try{
                              sleep(50);
                    }
                    catch(java.lang.InterruptedException e){
                              System.out.println("Excepcion en sleep:
"+e.getMessage());
                    }
```

HILO 8: "SubBridge".

En este apartado no se va a incluir código relativo a la implementación puesto que es un hilo más propio de la parte de comunicación, por lo tanto se hará más adelante. Sólo incidir en que este hilo existe puesto que en el lado del servidor en la parte de comunicación existe una operación que se bloquea a la espera de que le llegue información, y puesto que existe una interfaz gráfica de usuario y ésta no puede bloquearse, se introduce la lógica de la comunicación en un hilo para evitar el problema.

A continuación se muestra un diagrama de los hilos y su interacción con el bean.

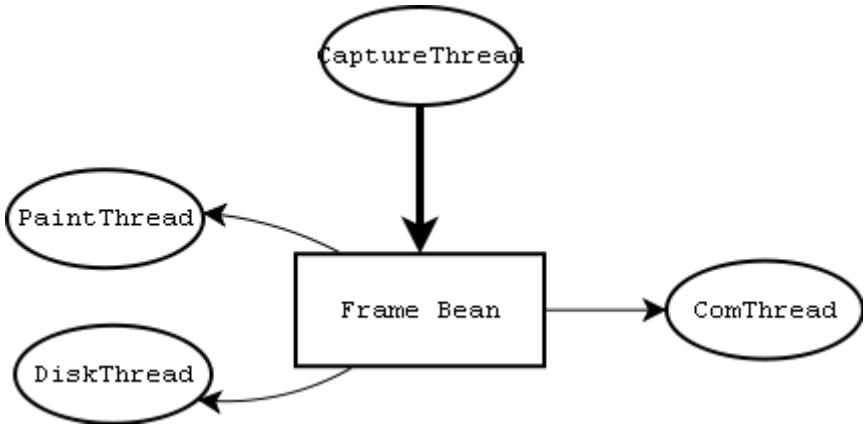


Figura 22: Esquema del acceso de los hilos al bean.

3.2.3 Diseño del Bridge Java-C++ y C++-Java

En este apartado se explica el problema más difícil de resolver en la creación de la aplicación de videovigilancia. Uno de los objetivos que se planteaba al inicio era el de probar el middleware **Ice** en un entorno heterogéneo en cuanto a lenguajes, tras lo cual se decidió implementar la parte cliente y servidor de usuario en **Java**, y la parte de comunicación en **C++.**

Bajo este criterio, se encontró el siguiente problema. Por un lado, en la parte cliente, el hilo **`CaptureThread`** tomaba la información de la cámara y la escribía en un bean **Java**, el cual era accedido por varios de los hilos de la aplicación que necesitaban dicha información. Entre esos hilos se encontraba **`ComThread`**, el cual era el encargado de enviar esa información al módulo en **C++** que era el "*publisher*" de la aplicación.

Con estas condiciones se posee en **`ComThread`** una estructura de datos en **Java**, más concretamente un *array* de bytes, que tiene que pasar a un programa externo en **C++**. El problema por tanto es, ¿cómo se puede compartir información entre dos programas independientes e implementados en distintos lenguajes de programación?

Entre las posibles soluciones, se baraja la posibilidad de integrar la aplicación de **C++** en la aplicación **Java** mediante la utilización de **JNI** (Java Native Interface), una característica de **Java** que permite realizar llamadas a métodos/funciones que están implementados en **C** o **C++.** Por tanto podría ser planteable realizar algo así. Otra solución, muchísimo menos eficiente, sería escribir el flujo de bytes que se tiene en el *array* a disco, y leerlo desde la aplicación **C++.** Esta solución por supuesto es inviable a todas luces, primero por el retardo que se introduciría en este punto de la aplicación, y segundo por el problema que supondría sincronizar ambas aplicaciones accediendo a un fichero externo a ambas.

Finalmente la solución propuesta es la más conveniente para el proyecto, puesto que es la propia utilización de **Ice** para realizar esta comunicación. En el capítulo del estado del arte de este proyecto se comentan las posibilidades del lenguaje **Slice**, que es el utilizado por **Ice**. Se trata del lenguaje en el cual es

necesario implementar las funciones, datos y demás construcciones que deseen compartirse entre dos puntos de la comunicación **Ice**. Su característica principal es que siendo un lenguaje independiente de plataforma y lenguaje de programación, permite precisamente a un cliente y a un servidor implementados en lenguajes distintos compartir una interfaz común, la cual puede contener estructuras de datos, llamadas a funciones, etc.

Con esta información, la solución no era otra que implementar un cliente **Ice** en **Java** que compartiese con un servidor en **C++** la información necesaria. Una vez hecho esto, se tendría el *array* de bytes de **Java** en una estructura en la aplicación **C++,** de este modo no habría más que enviarla al ordenador remoto. Una vez la información se encuentra en el ordenador remoto, se plantea nuevamente el mismo problema a la inversa. Se tiene una estructura de datos en **C++** que debe enviarse a una aplicación en **Java**, en forma de *array* de bytes, de modo que la aplicación en **Java** del servidor sea capaz de procesar dicha información para presentarla en disco. A estas construcciones en **Ice** se les ha llamado "*bridge*". Se tiene un bridge en el ordenador local que se encarga de pasar información desde una aplicación **Java** a una en **C++,** y un bridge en el ordenador remoto que se encarga de hacer justamente lo inverso.

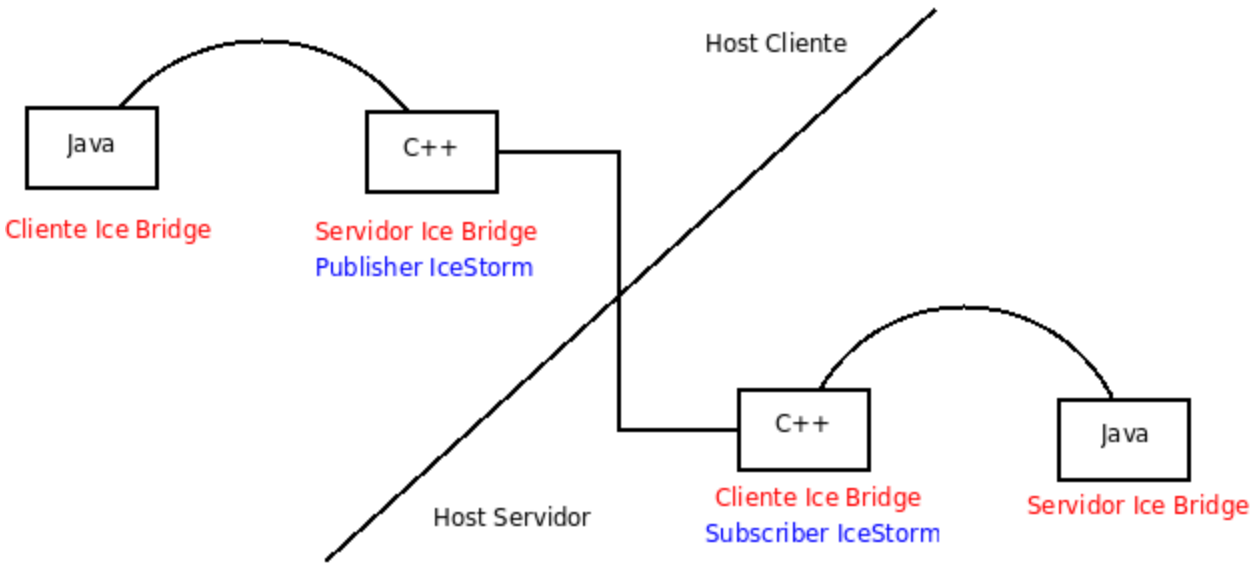


Figura 23: Esquema de la comunicación de la demo de videovigilancia. En rojo partes Java y en azul partes C++.

Es fácil pensar que una comunicación cliente-servidor pueda introducir mucho retraso en ambos bridges, pero no supone problema en el caso de **Ice**, puesto que cuando se configura cliente y servidor para operar en un mismo nodo, como es el caso de la aplicación de videovigilancia, la información se comparte utilizando el mecanismo de memoria compartida que ofrece el sistema operativo. De este modo el tiempo que se introduce es despreciable, incluso para una aplicación que requiere respuesta y rapidez como es el caso de la demo de videovigilancia, en la que se opera con flujos de vídeo.

Una vez descrita la solución que se propuso para el problema de la compartición de información, se pasa a describir la implementación, centrándonos en el funcionamiento de **Ice**, que además ayudará a comprender en el siguiente capítulo el funcionamiento de **IceStorm**, puesto que hay muchos pasos previos que se comparten.

En esta situación, lo primero de todo es definir una interfaz en **Slice**, para lo cual es necesario conocer cómo funciona el mapeado de **Java** a **Slice** y viceversa y de **C++** a **Slice** y viceversa, puesto que son necesarios tanto en el ordenador local como en el remoto. A continuación se muestra la definición de la estructura utilizada (sintaxis **Slice**).

```
module DataImage{
          sequence<byte> Data;
          interface Forward {
                    void report(Data imageData);
          };
};
```

Una estructura **`sequence<byte>`** de **Slice**, corresponde a un *array* de bytes en **Java**, precisamente la estructura en la que se dispone la información. Este es otro de los motivos por los que se decidió procesar la imagen capturada de la cámara para mantenerla en un *array* y no en un objeto **`ByteBuffer`**. Si se hubiera mantenido la información en un objeto **`ByteBuffer`**, posiblemente podría haberse realizado el mapeo con **Slice**, mediante técnicas de serializado y deserializado, pero esto obviamente hubiera sido mucho menos eficiente que la simple estructura que se muestra arriba.

Además se provee de una función, **`report(Data imageData)`** la cual es la encargada de redirigir los datos al siguiente punto, de este modo, en el cliente **Java**, se realiza un **`report(byte[]),`** que llegará al extremo **C++,** donde se recuperara un **`std::vector`**, que es la estructura de **C++** a la que se mapea un **`sequence<>`** de **Slice**.

Una vez vista la solución desde el plano teórico, se muestra a continuación la implementación de ésta, en la que se observará con detenimiento las líneas de código especialmente relevantes, que en este caso son las relacionadas con **Ice**.

En la parte del cliente, como se ha comentado, se debe enviar un *array* de bytes de la aplicación **Java** a la aplicación **C++,** por lo tanto en este caso, la parte del bridge perteneciente a la aplicación **Java** actuará como cliente de la comunicación, enviando los datos, y la parte de la aplicación **C++** involucrada en la comunicación actuará como servidor, recibiendo los datos.

Antes de mostrar el código de la implementación relativo a la parte cliente y servidora del bridge, es conveniente entender cómo funciona **Ice**, por tanto en la siguiente sección se explica dicho funcionamiento.

3.2.3.1 Funcionamiento Ice

El funcionamiento de **Ice** implica varios pasos, los cuales van a ser explicados a continuación, asociando cada uno al objeto/proceso que se usa en la implementación.

Objeto `Ice::Communicator`:

Supone el punto de entrada de toda comunicación en **Ice**. Es la base de la tecnología. A un objeto **`Communicator`** es posible asociar una serie de componentes con distintas funciones para la aplicación. De estos componentes hay

que destacar, el pool de hilos que tiene asociados, tanto si actúa como parte cliente, como si lo hace de parte servidora. Además, a este se asocian todos los adaptadores de objetos, que se verá más adelante qué son.

Communicator Initialization (Inicialización):

La inicialización del objeto **`Communicator`** es una operación obligatoria, que se realiza tras crear el objeto, y que deja a éste configurado conforme a los parámetros que se introduzcan sin permitir que se modifiquen estos una vez creado el objeto **`Communicator`**.

Object Adapter:

Cumple con varias funciones importantes. Entre las cuales se incluyen:

- Sirve como interfaz entre el código servidor de **Slice** (el que procesa las peticiones), y **Ice**.

- Permite que se proceda con el ciclo de vida de los objetos **Ice** y los "*servants*" para que en la creación y destrucción de éstos no se produzcan condiciones de carrera u otras situaciones no deseables.

- Provee al menos un "*endpoint*" (extremo), siendo "*endpoint*" un host:port. Los clientes acceden a los objetos **Ice** a través del "*endpoint*" que se configurara en el "*object adapter*".

A un "*object adapter"* están asociados varios "*servant*", los cuales a su vez estarán asociados a X objetos **Ice**. Para cada "*object adapter"* habrá configurado al menos uno, pero opcionalmente más "*endpoints*". Cada "*endpoint*" supone un camino alternativo para el cliente, que puede acceder a los distintos "*servants*" por medio de diferentes "*endpoints*". Por tanto este es un pequeño mecanismo para ofrecer redundancia. Ver figura 24.

Active Servant Map (ASM):

Cada "*object adapter"* tiene asociado un ASM, el cual supone un mapeo de los distintos objetos **Ice** que tenga asociados a una dirección de memoria o una referencia de un lenguaje concreto. De este modo el funcionamiento es el siguiente. Un cliente envía una petición a un servidor, a un "*endpoint*" concreto, como se ha dicho un "*endpoint*" corresponde a un único "*object adapter*", que estará asociado a un único objeto **`Communicator`** (que será el objeto **`Communicator`** de parte del servidor). Además de dirigirse a un "*endpoint"* concreto, el cliente incluye el identificador del "*servant*" que le interesa. De este modo cuando la petición llega al servidor, éste toma el nombre con el que ha realizado la petición el cliente, lo busca en el ASM y obtiene la referencia o la dirección de memoria donde se aloja el objeto en concreto.

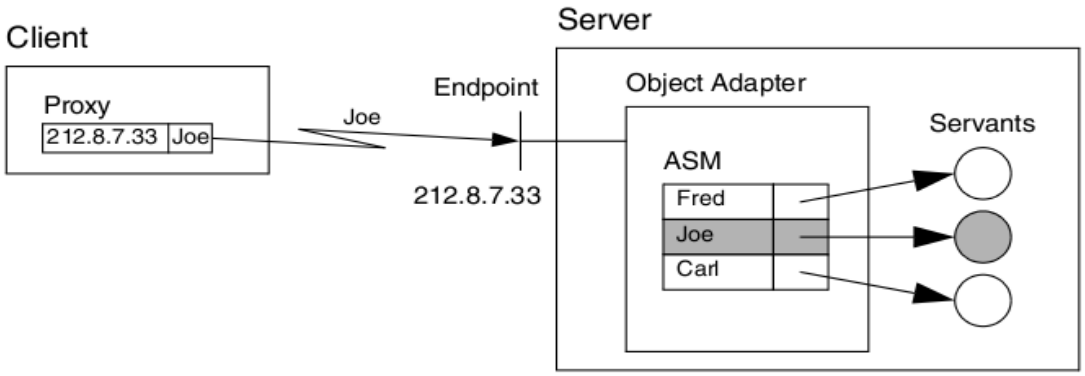


Figura 24: Esquema de funcionamiento de Ice.

Servant:

Un "*servant*", del que se ha venido hablando en las últimas líneas, consiste en la implementación que se encuentra en el lado del servidor para una operación dada. Es decir, cuando se realiza una petición en cliente, el resultado de esta será procesado y devuelto por el "*servant*" del lado de servidor, o dicho de otro modo, es la lógica que implementa cada función.

Servant Activation and Deactivation (Activación y desactivación):

La activación de un "*servant*" consiste en registrar éste en el ASM, de modo que sea accesible por los clientes. La desactivación es el proceso contrario. En el vocabulario del middleware, mientras un "*servant"* esta activado se dice que encarna el objeto **Ice**. Cuando se activa un "*servant*", se debe hacer con una identidad única, de modo que sea reconocible por esa identidad y no se puedan tener dos identidades iguales para dos objetos "*servant"* distintos en el mismo ASM, lo que provocaría errores graves. Las librerías de **Ice**, no obstante, incorporan funciones para asignar un identificador único a un "*servant*" dado.

3.2.3.2 Implementación de la solución

Tras haber visto un poco algunas características básicas del funcionamiento de **Ice**, se va a repasar el código relativo a la implementación de la solución, tanto en la parte Cliente (**`ComThread`**), como en la parte servidora (Bridge-Publisher). Se muestra a continuación el código relevante comentado.

```
//Los atributos de la clase referentes a la parte de la comunicación,
//imageData es el proxy de la comunicación, sobre el cual puede
//aplicarse llamadas al método report(), ic es un objeto
//Ice.Communicator el cual como se verá más adelante es la base de
//toda comunicación en el middleware Ice.

private DataImage.ForwardPrx imageData;
private Ice.Communicator ic;

//El constructor recibe por parámetro el bean desde el que tomara la
//información que necesite enviar, así como la variable goon encargada
//de controlar la ejecución de dicho hilo.

public ComThread(FrameBean dataUnit, boolean goon){
          this.dataUnit = dataUnit;
          this.goon = goon;
          ic = null;
```

```
                    try{

                    //En la construcción del hilo, se inicializa el objeto
                    //communicactor ic para poder operar.

                    ic = Ice.Util.initialize();

                    //Seguidamente creamos un proxy genérico de Ice, dándole
                    //la información pertinente, para saber donde encontrarlo,
                    //esto es host y puerto. Tenemos por tanto un proxy genérico
                    //base en localhost:10000.

                    Ice.ObjectPrx base =
ic.stringToProxy("DataImage:default -p 10000");


                    //Ahora forzamos a que el proxy se adapte a las condiciones
                    //de nuestro objeto concreto, el cual hemos definido con
                    //ayuda de Slice. De este modo, base pasa a ser imageData,
                    //el cual es un objeto proxy que escucha en localhost:10000,
                    //y que implementa el método report(DataImage imageData),
                    //siendo DataImage un sequence<byte> de Ice.

          imageData = DataImage.ForwardPrxHelper.checkedCast(base);


                    //Comprobamos que imageData no sea null, lo que
                    //significaría un error en la creación.

                    if(imageData == null) throw new Error("Invalid
proxy");
          }

          //Finalmente comprobamos las posibles excepciones que puedan
          //producirse en tiempo  de ejecución.

          catch (Ice.LocalException e){
                    System.out.println("Error Local en Ice:
"+e.getMessage());
          }
          catch (Exception e){
```

```
                    System.out.println("Error general en Ice:
"+e.getMessage());
          }
          }
```

En la parte servidora, la aplicación **C++,** se implementa lo siguiente.

```
//La clase ForwardI, definida en la parte servidora.

class ForwardI : public Forward {
          public:
                    //Se define el prototipo de la función report() como función
                    //virtual.

                    virtual void report(const DataImage::Data& data,
const Ice::Current&);
};

//Implementación del método report(). Por ahora la implementación concreta no
//interesa, solo es importante saber que esa implementación es lo que se ejecutara
//en cada llamada al método report() por parte del cliente.

void ForwardI::report(const DataImage::Data& data, const
Ice::Current&){
          [...]
}

//El método main() correspondiente a la implementación de la aplicación en C++.

int main(int argc, char* argv[]){

//Este método es el encargado de inicializar y obtener el publisher de la
//comunicación, pero esto se verá más adelante.

initPublisher();

          //Se crea un objeto communicator para poder establecer
          //comunicaciones.
```

```
            Ice::CommunicatorPtr comBridge;
            try{

            //Se inicializa ese objeto con ciertos parámetros de configuración que por
            //el momento son nulos.

            comBridge = Ice::initialize(argc, argv);

            //Se crea un adaptador en localhost:10000, al que puede asociarse
            //cierto proxy.

            Ice::ObjectAdapterPtr adapter = comBridge-
>createObjectAdapterWithEndpoints("DataImage", "default -p 10000");

            //Se crea un proxy basado en la interfaz ForwardI, definida más arriba, y
            //que implementa la misma definición que la propia de la parte cliente.

            Ice::ObjectPtr object = new ForwardI;

            //Se añade en este momento el proxy creado anteriormente al
            //adaptador, con la identidad "DataImage".

            adapter->add(object, comBridge-
>stringToIdentity("DataImage"));

            //Se activa el adaptador para que pueda operarse con el proxy
            //registrado.

             adapter->activate();

            //Método que bloquea la ejecución. En este momento el objeto del
            //adaptador (proxy) está activo y esperando llamadas, cuando recibe
            //llamadas se ejecuta la implementación del método report
            //correspondiente, y no termina hasta
            //que se detenga dicho proxy.

             comBridge->waitForShutdown();
}

            //Se capturan las excepciones susceptibles de saltar en ejecucion.
```

```
        catch (const Ice::Exception& e) {
                cerr << e << endl;
        }
        catch (const char* msg) {
                 cerr << msg << endl;
        }

        //Tras detenerse el proxy, el flujo de ejecución llega aquí, y si el proxy
        //comBridge no es null, entonces de destruye explícitamente para liberar
        //recursos.

        if(comBridge){
                try {
                        comBridge->destroy();
                }
                catch (const Ice::Exception& e) {
                        cerr << e << endl;
                 }
        }
}
```

3.2.4 Diseño de la Comunicación

El presente capítulo trata la parte más importante de la aplicación de prueba, la parte de comunicación entre nodos. En este apartado se mostrarán las características del entorno en el cual se realizará la comunicación, así como las decisiones tomadas durante el proceso de despliegue, el funcionamiento detallado del papel de **IceStorm** en esta aplicación y el código relevante que se ha implementado referente a **IceStorm**.

El planteamiento de la comunicación es el siguiente. Gracias a los bridges ("puentes software"), se puede mover la información de una aplicación en un lenguaje a otra sin problemas. En esta situación se tiene en el cliente una aplicación **C++,** que además de hacer de servidor **Ice** para recibir los datos, tendrá que enviar los datos al nodo remoto. Del mismo modo, se tiene en el servidor una aplicación en **C++** que aparte de ser cliente **Ice** para reenviar la información a la aplicación final, debe recibir la información del nodo origen.

Si se quiere buscar el mapeo de ésto a un paradigma de publicación-suscripción, la respuesta es rápida y sencilla. Se tiene por un lado una aplicación cliente en la cual se está generando información que debe llegar a otra aplicación servidora, por tanto esta aplicación tendrá el papel de publicador en la comunicación. En el otro lado se encuentra la aplicación servidora, que debe recibir la información que se publique, por tanto es claro que tendrá el papel de suscriptor. Además, en este contexto, el tópico ("*topic*") será un *frame* de la captura de la cámara.

Una vez claros los papeles que tendrá cada uno de los elementos de la aplicación, el siguiente paso es ver cómo funciona **IceStorm**, el servicio de **Ice** que nos permitirá desplegar un contexto de publicación-suscripción. A entender el funcionamiento de **IceStorm** se destina el siguiente apartado.

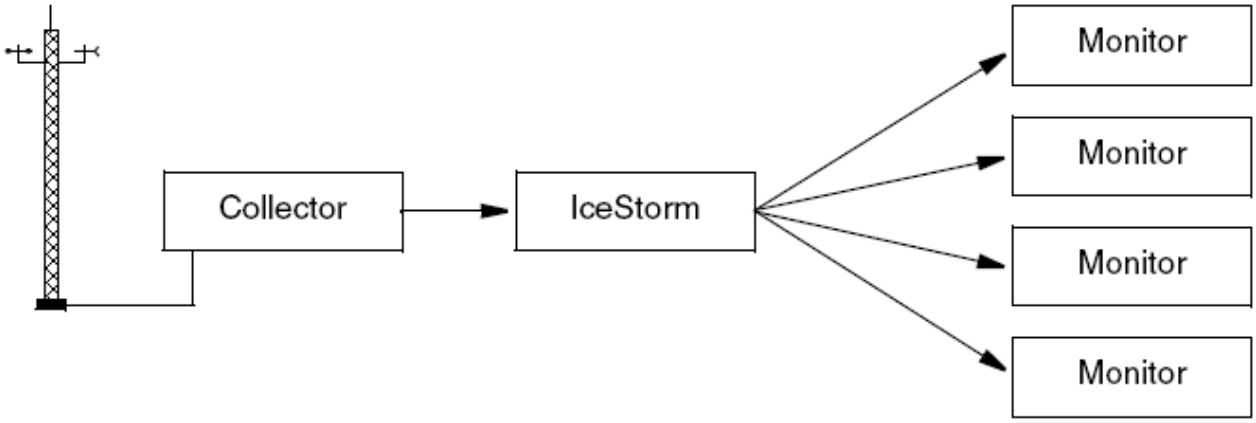


Figura 25: Ejemplo aplicación IceStorm.

3.2.4.1 Funcionamiento de IceStorm

En este apartado se procede como en el 3.2.3.1, en el que se explico **Ice**, pero centrándonos en las diferencias que se encuentran al utilizar el servicio especifico de **Ice**, **IceStorm**.

Para empezar, se hace necesario conocer los términos utilizados para nombrar ciertos conceptos de importancia para **IceStorm**.

IceStorm Message ("mensaje IceStorm"):

Un mensaje **IceStorm** representa la llamada a una operación de **Slice**. El nombre del mensaje será el nombre de la operación, y el contenido del mensaje son los parámetros que incluya la operación. Un mensaje se publica realizando la llamada a la operación deseada sobre el proxy del cliente, y este mensaje se recibirá por una llamada que se hará al "*servant*" de la parte servidora, que será el "*subscriber*" (suscriptor), es decir exactamente del mismo modo que en el caso de una comunicación con **Ice**.

Topic (“tópico”):

El “*subscriber*” debe registrar aquellos tópicos por los que tenga interés, de modo que el servicio **IceStorm** sepa que datos publicados (por un número de publicadores) son de interés de un “*subscriber*” concreto.

Un tópico es equivalente a la interfaz definida en **Slice**, de modo que un cliente puede operar con las operaciones definidas en esa interfaz, que será la interfaz del tópico, y un servidor debe implementar esa interfaz, que será la implementación del tópico. Este comportamiento como se comprueba es el mismo que en una comunicación normal con **Ice**, la diferencia fundamental es que en ningún momento se presta atención para definir “*endpoints*”, sino que simplemente se tiene un cierto número de “*publishers*” que publican información al servicio **IceStorm**, y este reenvía esa información a los “*subscribers*” que hayan presentado su interés por dichos tópicos (“*topic*”).

Unidirectional Messages (“mensajes unidireccionales”):

En **IceStorm** los mensajes son siempre unidireccionales, un publicador realiza una llamada sobre una función cuyo valor de retorno siempre será **`void`**, no hay respuesta posible de los suscriptores a los publicadores.

Quality of Service (“Calidad de servicio”):

IceStorm tiene dos parámetros de calidad de servicio que pueden configurarse para conseguir alguna funcionalidad o algún comportamiento por parte de la aplicación, más determinado. Esto es más comentado en el capítulo 2, el estado del arte. Es quizá uno de los puntos débiles de **IceStorm**, ya que ambos parámetros son muy útiles, pero se echa en falta algunos más que pudieran ofrecer una mayor granularidad a la hora de configurar nuestra aplicación para que se comporte de una forma determinada, y de este modo no queda otro remedio más que solucionar tales deficiencias de forma programática.

Persistent Mode ("Modo persistente"):

El modo por defecto de funcionamiento del servicio **IceStorm** es persistente, es decir que mantiene guardados en una base de datos información relativa al "*topic*" y los "*subscribers*" de dicho "*topic*", entre otras cosas, en una base de datos. En todo caso, los mensajes concretos no se guardan, son entregados al "*subscriber*" y no se mantienen en ningún lugar de forma persistente.

Transient Mode ("Modo transitorio"):

Para consumir menos recursos y no depender de un dispositivo de memoria persistente donde poder almacenar la información que genera el servicio **IceStorm**, éste provee opcionalmente un modo en el que ninguna información se guarda en ninguna base de datos.

TopicManager:

Es el objeto base del servicio **IceStorm**, viene a realizar un papel similar al **Active Servant Map** en **Ice**. En un **TopicManager** se registran una serie de tópicos. Tanto publicador como suscriptor deben acceder al **TopicManager** que reside en la máquina donde se encuentre el servicio **IceStorm** para tomar un proxy asociado a dicho tópico en el caso de un publicador, o para suscribirse a cierto tópico y obtener un proxy asociado a dicha comunicación en el caso de un suscriptor.

Una vez visto algunos puntos clave de **IceStorm** se ofrece la implementación del "*publisher*" y del "*subscriber*" en la aplicación, con la explicación de cada uno de los pasos dados en el código.

3.2.4.2 Implementación de la comunicación IceStorm

En la parte del "*publisher*" se inicializa éste del siguiente modo:

```
//El método initPublisher es llamado desde el main de la aplicación, antes de
//configurar el proxy para el bridge, como se puede comprobar en el código del
//bridge comentado más arriba.

void initPublisher(){

        //Como siempre en Ice, lo primero es crear un objeto Communicator e
        //inicializarlo.

        Ice::CommunicatorPtr comPublisher;
        comPublisher = Ice::initialize();

        //Tras esto asociamos este objeto Communicator al host donde reside el
        //servicio IceStorm, que será más concretamente donde se encuentre el
        //TopicManager que nos interesa, y hacemos un casting para obtener
        //ese TopicManager.

        Ice::ObjectPrx obj = comPublisher-
>stringToProxy("IceStorm/TopicManager:tcp -h 163.117.141.54 -p
10010");
        IceStorm::TopicManagerPrx topicManager =
        IceStorm::TopicManagerPrx::checkedCast(obj);

        //El siguiente paso es tomar o crear el objeto topic del topicManager,
        //para poder obtener más tarde un objeto publisher asociado a tal topic.

        IceStorm::TopicPrx topic;
        try{

                //Intentamos tomar el objeto topic con nombre ImageData del
                //topicManager.
```

```
                topic = topicManager->retrieve("ImageData");
        }
        catch(const IceStorm::NoSuchTopic&) {

                //Si el intento anterior falla, entonces creamos ese topic.

                topic = topicManager->create("ImageData");
        }

        //Una vez tenemos el topic sobre el que queremos publicar, es necesario
        //que obtengamos un proxy para poder hacerlo, y para esto aplicamos la
        //function getPublisher() sobre el topic obtenido.

        Ice::ObjectPrx pub = topic->getPublisher()->Ice_oneway();

        //Finalmente guardamos el casting del publisher obtenido a través del
        //topic en forward, que es un variable global de tipo ForwardPrx, es decir,
        //es el proxy sobre el que podemos operar con las operaciones definidas
        //en la interfaz Slice.

        forward = ForwardPrx::uncheckedCast(pub);
}
```

Ahora es importante realizar una serie de reflexiones. Por un lado se tiene del lado del cliente en la aplicación **C++**, la implementación del servidor del bridge, y la implementación del "*publisher*". El fin del bridge no es otro que ofrecer al "*Publisher"* los datos de la cámara en un **`std::vector`**, para que este "p*ublisher"* pueda publicarlos.

Por otro lado se puede observar que puede utilizarse la misma interfaz **Slice** de los "bridge" para realizar la comunicación entre nodos, aumentando la eficiencia de la aplicación, puesto que tienen que publicarse unos datos, exactamente igual que en el caso del bridge, no hay más operaciones.

Con estas dos cosas claras, se converge en una solución, que es utilizar el proxy del "*publisher*" (*forward*) para publicar los datos que hayan llegado en la implementación del método **`report().`**

Para esclarecer un poco esto se muestra a continuación el proceso completo desde que se genera un *frame*, hasta que este es publicado por el "*publisher*", acompañado de la siguiente imagen.

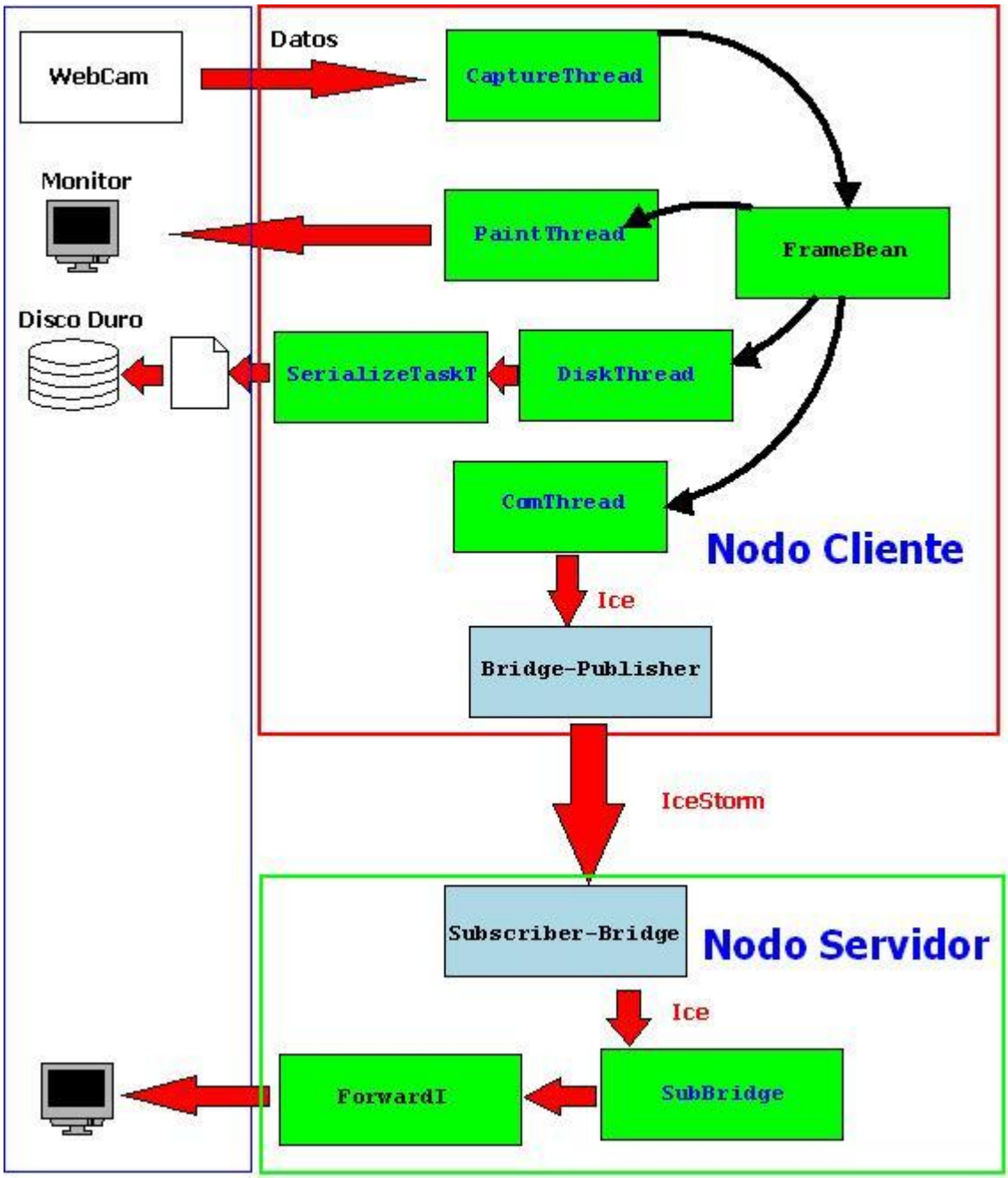


Figura 26: Esquema funcionamiento aplicación.

En la ilustración anterior encontramos tres grandes bloques. El bloque de la izquierda representa el hardware implicado en el funcionamiento de la aplicación. El cuadro superior, de borde rojo, incluye los elementos del nodo cliente. El cuadro inferior, de borde verde, incluye los elementos del nodo servidor. Aparece en texto azul los hilos, y en negro los elementos o clases normales. Con fondo verde las clases **Java**, y en fondo azul claro las clases **C++.**

En la comunicación se ven implicados, **`CaptureThread`**, **`ComThread`**, toda la parte **C++,** y **`SubBridge`**.

1-. (Cliente-Java) El hilo **`CaptureThread`** toma un *frame* (**`byte[]`**) de la cámara y lo guarda en el bean correspondiente.

2-. (Cliente-Java) El hilo **`ComThread`** toma un *frame* (**`byte[]`**) del bean y lo envía a la aplicación Cliente-C++. Para esto hace uso del bridge.

2.1-. **`ComThread`** implementa un cliente **Ice**, del que consigue finalmente un proxy, el cual usa para enviar los datos (**`byte[]`**).

```
imageData = DataImage.ForwardPrxHelper.checkedCast(base);
imageData.report(data);
```

3-. (Client-C++) La aplicación Bridge-Publisher, implementa un servidor **Ice** para recibir los datos que ha enviado el cliente **Ice**. Los recibe en un formato manejable por el lenguaje **C++** gracias a la interfaz independiente de lenguaje definida en **Slice**.

4-. (Client-C++) Implementa el "*servant*" del bridge, por tanto implementa el método **`report().`** Esta aplicación lo que debe hacer con los datos es publicarlos gracias al "*publisher*" que crea, por tanto eso es lo que se hace con los datos.

4.1-. La implementación del método **`report()`** es la siguiente:

```
void ForwardI::report(const DataImage::Data& data, const
Ice::Current&){
        forward->report(data);
}
```

Siendo forward el proxy del publicador, que ha sido inicializado y correctamente obtenido antes de que el servidor de **Ice** se bloquee esperando llamadas.

5-. De este modo, cuando el cliente **Ice** (Cliente-Java) envía algo al servidor **Ice** (Cliente-**C++**), lo que se hace con los datos que llegan es pasárselos al método **`report()`** nuevamente, pero esta vez es invocado por el publicador de la aplicación.

Tras observar el recorrido de la información hasta que ésta es publicada por el publicador de la comunicación, se pasa a comentar el funcionamiento del suscriptor, y nuevamente el ciclo que sigue la información hasta que llega a la aplicación final.

Por tanto a continuación se muestra la implementación relativa al suscriptor de la aplicación (Servidor-**C++**).

```
//El método main dela aplicación.

int main(int argc, char* argv[]){

        //Método encargado de iniciar el cliente Ice para enviar los datos
        //recibidos por el subscriber a la aplicación final. Se ve más adelante.

        initBridge();
```

```
		//Objeto base de la comunicación e inicialización de este.

		Ice::CommunicatorPtr comSubscriber;
		 comSubscriber = Ice::initialize(argc, argv);

		//Asociamos el objeto Communicator al host donde corre el servicio
		//IceStorm, para acceder al objeto TopicManager

		Ice::ObjectPrx obj = comSubscriber-
>stringToProxy("IceStorm/TopicManager:tcp -h 163.117.141.54 -p
10010");
		IceStorm::TopicManagerPrx topicManager =
IceStorm::TopicManagerPrx::checkedCast(obj);

		//Ahora creamos un object adapter sobre el cual registraremos el
		//servant, de modo que las peticiones entrantes puedan ser
		//despachadas.

		Ice::ObjectAdapterPtr adapter = comSubscriber-
>createObjectAdapterWithEndpoints("ImageDataAdapter", "tcp:udp");

		//Creamos el objeto Ice que implementa la interfaz definida en Slice.

		ForwardPtr forward = new ForwardI;

		//Obtenemos el proxy, añadiendo con una identidad unica el objeto Ice al
		//objectApapter.

		Ice::ObjectPrx proxy = adapter->addWithUUID(forward)-
>Ice_oneway();

		//Creamos un objeto topic, que recuperaremos tomandolo del
		//topicManager, del mismo modo que en el caso del publisher.

		IceStorm::TopicPrx topic;
		try {
			topic = topicManager->retrieve("ImageData");

			//Definimos unos parametros de calidad de servicio, que en
```

```
                    //este caso no son configurados.

                    IceStorm::QoS qos;
                    //Asociamos el proxy y los parámetros de calidad de servicio
                    //al topic.

                    topic->subscribeAndGetPublisher(qos, proxy);
          }
          catch (const IceStorm::NoSuchTopic&) {
                    cout << "Excepcion al suscribirse a un topico."
<< endl;
          }

          //Activamos el objectAdapter para que sea visible.

          adapter->activate();

          //Bloqueamos el flujo de ejecución hasta que se detenga el Subscriber.

          comSubscriber->waitForShutdown();

          //Una vez detenido desasociamos el proxy del topic.

          topic->unsubscribe(proxy);
}
```

Tras observar la implementación del "*subscriber*", se repasa el ciclo que sigue la información una vez es publicada por un cliente.

1-. (Cliente-C++) El "*publisher*" de la comunicación ha publicado información en un tópico para el cual el "*subscriber*" está interesado.

2-. (Servidor-C++) **IceStorm** sabe esto y por tanto le pasa esta información, por supuesto esto se realiza con el método **`report()`** que es el usado por el "*publisher*". De esta forma el "*subscriber*" debe implementar el método **`report()`** para operar con dicha información.

2.1-. (Servidor-C++)La interfaz para el método **`report()`** en el "*subscriber*" es la siguiente:

```
void ForwardI::report(const DataImage::Data& data, const
Ice::Current&){
        forwardBridge->report(data);
}
```

Como se aprecia el "*subscriber*", una vez recibe la información, utiliza el bridge (**`forwardBridge`**), que ha configurado y obtenido previamente, para reenviar la información a la aplicación final. El proxy **`forwardBridge`** es configurado cuando se realiza la llamada a la función **`initBridge()`** en el método **`main()`** de la aplicación.

El funcionamiento del lado servidor guarda cierta simetría con el lado cliente, aunque los papeles de servidores y clientes **Ice** se implementan en lenguajes distintos, lo que ayuda a observar el comportamiento de la tecnología en ambos lenguajes.

3.3 Resumen

En este capítulo se han mostrado los principales problemas que ha supuesto la realización de la aplicación y los pasos seguidos para su resolución. Además se ha hecho hincapié en el funcionamiento de la tecnología **Ice**.

Se han revisado los drivers y la tecnología utilizada para resolver el problema de acceso a la webcam y recuperación de información a través de ésta. **V4L4J** ha sido el API utilizado por ofrecer la funcionalidad de los drivers nativos **V4L** en el lenguaje de programación **Java**.

Con la parte cliente resuelta en lenguaje **Java**, se ha cumplido con el requisito de que la comunicación entre los nodos remotos se realizase con una implementación en el lenguaje **C++**. Para esto, como se ha visto, se han tenido que desarrollar dos bridges, para poder compartir la información entre las aplicaciones **Java** y **C++** del lado del cliente y las propias entre **C++** y **Java** de lado del servidor. Estos bridges se han diseñado como comunicaciones **Ice**, aprovechando de esta forma la potencia del lenguaje **Slice** para definir el formato de los datos que se tienen que compartir y obteniendo los correspondientes mapeos en la parte **Java** y en la parte **C++**. Se ha comprobado en este punto que **Ice** está bien optimizado para operar en memoria compartida, en host local, y ha significado una ausencia de retardo significante en estos puntos de la comunicación.

Tras observar cómo se resolvía el problema de los bridges, se afrontó la comunicación entre ambos nodos por medio de la utilización del servicio **IceStorm**. Visto **Ice**, utilizar **IceStorm** es tener en cuenta un par de conceptos nuevos y saber identificar bien los distintos componentes que conformarán la comunicación del paradigma de publicación-suscripción.

Finalmente tras todos los desarrollos comentados se tienen dos aplicaciones por parte de cliente y otras dos por parte de servidor. Cada una implementada en un lenguaje distinto, **Java** y **C++** respectivamente. Las aplicaciones más externas, con las que un usuario puede interactuar, ya sea el control en la parte cliente, o la visualización en la parte servidora, están escritos en

Java. Por otro lado, las partes más internas de la aplicación, encargadas de comunicar un nodo con el otro, están escritas en **C++**. Cada par de aplicaciones se comunica mediante los bridges de los que se ha hablado, y el conjunto ofrece una solución para la demo de videovigilancia remota que se propuso.

Capítulo 4: Estudio y Evaluación de IceStorm.

4.1. Introducción y Objetivos

Como se comentaba al inicio del primer capítulo, la decisión a la hora de elegir un middleware para una aplicación concreta es delicada, debido a la gran importancia que tiene, puesto que la elección de un middleware u otro puede determinar el comportamiento de la aplicación final.

Además, está claro que no todos los middleware ofrecen lo mismo, y que no todas las aplicaciones tienen los mismos requisitos, por tanto un middleware concreto puede ser el más apto para una aplicación de vídeo, y quedarse muy corto para una aplicación crítica de tiempo real que vaya a ir embarcada en aeronaves, etc.

Delimitando objetivos, tras el capítulo del estado del arte queda claro que **Ice** es un middleware genérico, que no está optimizado para aplicaciones críticas, partiendo del simple hecho de que no está preparado para aplicaciones con restricciones de tiempo real. Además, la particularización que tiene para sistemas embarcados, **Ice**-**E**, debe entenderse como un esfuerzo para ofrecer la funcionalidad de **Ice** a dispositivos con bajas capacidades, como puedan ser teléfonos móviles, PDA's, etc., pero nuevamente no es una tecnología preparada para nodos limitados como sensores y actuadores en un entorno de tiempo real crítico.

Hay que tener claro entonces que se deberá tener en cuenta el middleware **Ice** cuando la aplicación para la cual vaya a ser utilizado se trate de una aplicación de propósito general, para entornos con recursos limitados. **Ice** en este sentido ofrece una huella en memoria considerablemente baja. Esto por otro lado no es para nada un inconveniente, las tecnologías se desarrollan con unos objetivos y unas metas, y en el caso de **Ice** está claro que no perseguía ser apto para aplicaciones críticas, puesto que como se ha repetido, ni implementa características de tiempo real, ni tiene soporte para sistemas operativos de tiempo real, ni tiene "*mappings"* para lenguajes que suelen usarse en dicho entorno, como pueda ser Ada por ejemplo.

Teniendo claro entonces que el middleware sobre el que se ha realizado el estudio es un middleware orientado a aplicaciones de propósito general, y con bastantes facilidades para utilizarse en aplicaciones orientadas a servicios web, como puede intuirse tras ver que existen "*mappings"* para lenguajes como **PHP**, **Ruby**, **Python**, o **Java**, deben medirse las posibilidades de dicho middleware en función a los fines para los que está desarrollado.

Por esto último, es necesario definir qué otras tecnologías middleware están desarrolladas para competir en el mismo terreno que **Ice**, y éstas son en principio **Java RMI**, y **Windows WCF**. Se deja de lado por tanto **DDS**, ya que si bien se ha realizado una extensa comparación de ambas tecnologías en el capítulo 2, el estado del arte, ésta ha servido para ver precisamente que son dos tecnologías enfocadas a distintos segmentos del mercado e incomparables en una aplicación real.

El fin de este capítulo no es otro que ofrecer algunas medidas adicionales a las expuestas por el desarrollador, ZeroC, de modo que si fuera necesario plantear la utilización de un middleware u otro, siempre en el ámbito de aplicaciones de propósito general, se posea un material adicional para tomar tal decisión.

Por tanto, durante el capítulo se va a ofrecer una descripción de las aplicaciones que se han implementado para realizar las mediciones en cuestión, en qué se basan las mediciones y qué fin persiguen éstas, y finalmente ofrecer los resultados de dichas mediciones.

4.2. Herramientas y Descripción

En esta sección se presentan las aplicaciones que se han utilizado para realizar las mediciones, pero antes es importante saber qué se ha pretendido medir, para entender por qué las herramientas utilizadas son como son.

A la hora de plantear la utilización de un middleware, la primera característica que tiene que atenderse y la cual limita la posibilidad o no de usarlo es que dicho middleware cubra las necesidades que perseguimos. En el caso de **Ice** se tiene un middleware orientado a objetos, muy preparado además para orientarlo a servicios web, con una extensión para dispositivos limitados, y lo más importante, con una serie de servicios que le dotan de una versatilidad muy importante. Como es normal, dada la línea que sigue este proyecto, el servicio **IceStorm**, que permite plantear una arquitectura basada en el paradigma de publicación-suscripción es sin duda uno de los puntos fuertes de **Ice**.

Ice, por tanto, nos ofrece una larga lista de posibilidades, de hecho, ofrece en general lo que puede llegar a ofrecer **CORBA**, que ha sido el middleware por excelencia durante muchos años, lo que nos da una idea de la cantidad de mercado que puede abarcar. Pero además de la amplitud de aplicaciones para las que **Ice** pueda encajar, otro de los puntos clave que suelen valorarse es la rapidez de desarrollo, puesto que este tiempo suele suponer costes, etc. Y en este punto desde luego **Ice** dispone de una de sus mejores bazas. Primero por ofrecer una amplia gama de "*mappings*" a los lenguajes de propósito general más utilizados en la actualidad, y a alguno más enfocado a la ejecución en servidor, como pueda ser **PHP**, o quizá **Ruby**. Y segundo, por disponer de una buena base de documentación, unido a la inherente sencillez de la tecnología en sus usos más básicos.

Figura 27: Algunos lenguajes soportados por Ice.

Lo anterior, unido a las facilidades que proveen los servicios, supone por tanto otra característica estrella de **Ice**. Los servicios permiten adaptar el middleware a topologías en las que existen firewalls hardware por ejemplo, o permiten registrar toda la actividad del sistema en una base de datos, etc.

Paralelamente a las anteriores características, e independientemente de ellas, existen ocasiones en las cuales no puede permitirse la elección de aquel middleware cuyo desarrollo facilite más las cosas, sino que existen ciertas características de rendimiento que debe cumplir para poder usarse, transformándose estas características en condiciones necesarias, y por tanto delimitando demás facilidades.

En este capítulo precisamente se tratará de mostrar algunos aspectos de **IceStorm** en su rendimiento, de modo que si fuera necesario atender a tales características sirva como una pequeña guía que complemente a los datos ofrecidos por ZeroC.

Con este objetivo, se plantean varias mediciones a realizar sobre el middleware **IceStorm**, que buscan una serie de objetivos concretos. Por supuesto estas pruebas dependen del hardware utilizado y tienen también mucha relación con el sistema operativo sobre el que se realizan, por tanto, se intentarán interpretar los resultados en función de estos factores “externos”.

Las mediciones concretas y el objetivo que persiguen se detallan en la sección siguiente (4.3), a continuación se muestra la estructura de las aplicaciones desarrolladas para llevar a cabo dichas mediciones.

Se parte de la idea de que se evaluará **IceStorm** con respecto a dos lenguajes, **Java** y **C++**, los más potentes de entre los que soporta. Y con esta idea se desarrollan tres aplicaciones.

Primero se comentan las dos primeras, que pretenden ser lo más similares posibles entre sí, salvaguardando el hecho de que estén escritas en distinto lenguaje. Los lenguajes son bastante diferentes si nos referimos a la forma de ejecución que tienen ambos. Esto es una restricción que se conoce desde el

inicio, y es por ello que las aplicaciones están desarrolladas del modo más minimalista posible, introduciendo el menor número de sentencias inservibles o redundantes en cada uno de los dos lenguajes. De este modo las aplicaciones de prueba se han intentado adaptar lo más posible a lo que ofrece el lenguaje "por defecto", perjudicando lo menos posible la ejecución de ambos.

El objetivo de estas dos primeras aplicaciones es realizar dos tipos de pruebas. Ambas aplicaciones miden **IceStorm** desde el lado del "*publisher*". Por un lado una prueba consiste en medir el tiempo de envío, que no es más que la suma del tiempo que tarda el sistema en serializar los datos y el tiempo que tarda en sacarlos por la interfaz de red, a esta prueba se le ha llamado **Send Time**. La otra prueba consiste en medir el tiempo de realización de 10000 envíos, lo cual tiene un matiz con respecto a la prueba anterior, y es que en este caso y suponiendo presencia de interferencias, los cambios de contexto que se produzcan en el sistema operativo delimitarán de un modo u otro el funcionamiento de **IceStorm**, por tanto se pretende observar la robustez de la tecnología frente a este hecho. A esta prueba se le ha llamado **Total Time**.

El hecho de implementar dichas aplicaciones en dos lenguajes diferentes tiene como objetivo comparar ambos lenguajes con respecto a su uso en la tecnología que tratamos.

Observando estas aplicaciones desde un punto de vista arquitectónico, las aplicaciones constan de dos partes, el "*publisher*" y el "*subscriber*". El "*publisher*" recibe por línea de comandos dos parámetros, un número y una palabra. El número hace referencia al tamaño del mensaje que se enviará, y la palabra (free, cpu o mem) hace referencia al tipo de interferencia que va a introducirse en tiempo de ejecución.

De esta forma puede decirse que la aplicación es configurable con respecto a dos aspectos, la cantidad de información que va a enviar ("*payload*"), y la interferencia que se va a simular en el sistema, que son los dos parámetros sobre los que se realizarán las comparaciones, mediciones, etc.

Además, se introducen mecanismos para controlar el tiempo que lleva la ejecución de ciertos segmentos del código, para lo cual se han utilizado librerías de

los dos respectivos lenguajes.

A continuación se presenta el algoritmo que siguen estas dos primeras aplicaciones que estamos tratando, el código concreto de cada una de las dos aplicaciones se ofrece al final del documento en un apéndice.

Publicador ("Publisher"):

- Establecimiento de la comunicación **Ice** con un "*endpoint"* por defecto.
- Registro o recuperación del tópico **`Data`**.
- Comprobación de argumentos de línea de comandos.
- Generación de vector con un número de bytes, el cual es indicado por el parámetro introducido por línea de comandos.
- Comprobar segundo parámetro y arrancar hilo de ejecución con interferencia de memoria (mem) o con interferencia de cpu (cpu), o sin interferencia (free).
- Configuración mecanismos para medir tiempo.
- Enviar datos.
- Comprobar tiempo y escribir datos en la salida estándar.
- Tras tomar la medida, continuar enviando datos indefinidamente o hasta completar un total de 10000 envíos.

Suscriptor ("Subscriber"):

- Establecer comunicación con el mismo "*endpoint"* que el "*publisher"*.
- Arrancar el "*subscriber"* y esperar peticiones.
- El "*servant"* ejecuta el código.

Observaciones:

Para conseguir la interferencia de cpu, el hilo encargado de ejecutar el código asociado a dicha interferencia se encarga de abrir un fichero en el path `/dev/null`, y escribir una cadena de texto dada (común para la aplicación **C++** y **Java**) indefinidamente. De este modo se consigue aumentar la ejecución de CPU

hasta el límite, puesto que supone una cadena infinita de operaciones de entrada a un sistema de ficheros, pero por otro lado, ya que el fichero especial /dev/null descarta todo aquello que se le redirecciona, no se compromete la estabilidad del sistema.

Por otro lado la interferencia de memoria se realiza de un modo bastante sencillo. Cuando se inicia este hilo, se crean varios vectores o arrays (dependiendo el lenguaje), con un tamaño ajustado empíricamente para conseguir que cada una de las aplicaciones fuerce el sistema en cuanto a memoria se refiere.

Aplicación RTT:

Tras describir los objetivos y la funcionalidad que implementan las dos primeras aplicaciones, se pasa a tratar la tercera aplicación desarrollada. Ésta mantiene las variables de medición, es decir, el "*payload*", y la interferencia, pero el objetivo no es medir sólo el comportamiento del "*publisher*", sino de la infraestructura **IceStorm** al completo, por ello la interferencia, en caso de existir, lo hará tanto en la parte cliente como en la parte servidora.

Para conseguir este fin se plantea una aplicación que mide el tiempo que transcurre desde que se envía un paquete, con un determinado "*payload*" desde un nodo a otro, y este mismo paquete vuelve al primer nodo, es decir el RTT (*Round Trip Time*). Esta aplicación al contrario que las dos primeras sólo esta implementada en el lenguaje **C++,** ya que como se verá en la siguiente sección es el que más rendimiento consigue de **IceStorm**.

Estructuralmente hablando, la aplicación consta de dos publicadores y dos suscriptores, con dos tópicos respectivamente. El primer publicador utiliza un tópico ("**`Data`**") para enviar los datos al suscriptor del nodo remoto, y éste, al recibir éstos, utiliza otro tópico distinto ("**`DataRETURN`**") para enviar dichos datos de vuelta al emisor original.

A continuación, y del mismo modo que anteriormente, se ofrece el algoritmo de la aplicación, y se incluye el código de la misma en un apéndice al final del documento.

Nodo A:

- Establecimiento de la comunicación Ice.
- Se consigue un publicador asociado al tópico **`Data`**.
- Se configura un suscriptor asociado al tópico **`DataRETURN`**. La funcionalidad de este suscriptor se encuentra en un hilo de ejecución aparte del inicial.
- Se toman parámetros por línea de comandos y se construyen los datos y se arranca la interferencia correspondiente, que se ejecuta en un hilo aparte.
- Se guarda el tiempo actual y se envía el dato al suscriptor asociado al tópico **`Data`**.

Nodo B:

- Se establece un suscriptor en el hilo principal de ejecución de la aplicación, asociado al tópico **`Data`**.
- Se establece un publicador asociado al tópico **`DataRETURN`**.
- Se reciben datos del tópico **`Data`**, estos se envían utilizando el publicador asociado al tópico **`DataRETURN`**.

Nodo A:

- El suscriptor asociado al tópico **`DataRETURN`**, detecta los datos devueltos, se registra el tiempo y se halla la diferencia, consiguiendo por tanto el tiempo de ida y vuelta.
- Una vez registrado el tiempo, se envían nuevamente los datos, con el publicador asociado al tópico **`Data`**.

La misma aplicación anterior, se utiliza para realizar la última prueba del estudio, que es la trata el RTT en localhost, es decir, se realiza el envío y vuelta en el mismo nodo, de modo que los datos se compartan por memoria compartida. Esta prueba tiene como fin ver el retardo que introduce la red en la comunicación anterior.

Tras repasar las aplicaciones desarrolladas nos encontramos finalmente con 6 pruebas.

- Java Send Time
- C++ Send Time
- Java Total Time
- C++ Total Time
- RTT localhost
- RTT

4.3. Resultados del estudio

En esta sección se presentan las consideraciones que se han tomado a la hora de decidir qué parámetros de **IceStorm** iban a medirse, así como la justificación de éstas. Del mismo modo se explicarán algunos detalles más concretos en cuanto a los mecanismos utilizados a la hora de implementar la idea generada. Más tarde se mostrarán finalmente los resultados que se han obtenido, y se realizará una comparación entre ellos.

Para la sección final del capítulo se dejan las conclusiones que pueden obtenerse a raíz de los datos presentados en esta sección.

4.3.1. Consideraciones para las medidas

Del middleware se espera que tenga un comportamiento lo más uniforme posible durante todo el funcionamiento de la aplicación, puesto que si ésta depende activamente del rendimiento del middleware, como es el caso de aplicaciones que comparten datos de forma distribuida, y el middleware no mantiene un comportamiento determinista, el rendimiento de la aplicación presentará picos de rendimiento, que en muchos casos pueden suponer un mal funcionamiento temporal de la aplicación o convertir la aplicación en algo no aceptable por el usuario.

Concretamente, se espera que un cambio en la utilización de recursos de la aplicación principal no interfiera demasiado en el funcionamiento del middleware. Es decir, si la aplicación del usuario hace uso de una cantidad mayor de recursos durante un tiempo determinado, lo más deseable es que el funcionamiento del middleware se vea afectado en la menor medida posible.

Es deseable en entornos donde el rendimiento es importante, conocer el comportamiento del middleware bajo situaciones con un consumo de memoria especial o un consumo de ciclos de cpu elevado. El problema entonces es conocer

cómo se comporta el middleware ante tales situaciones, puesto que los fabricantes dan a menudo unos resultados un tanto sesgados o directamente no ofrecen toda la información posible.

Con este pretexto, se han realizado las tres aplicaciones de las que se habla en la sección anterior (4.2). El fin es medir el comportamiento de **Ice** variando el tamaño del "*payload*" (carga de datos) que debe enviar, para ver con qué cantidad de información funciona mejor, y ver también cómo varía en función de una interferencia de memoria y una de cpu, tanto desde el punto de vista del "*publisher*", en el caso de las dos primeras aplicaciones, como desde la tecnología en general, en el caso de la tercera aplicación.

Para realizar los contrastes y las comparaciones es necesario tomar medidas cuantitativas, de modo que puedan usarse herramientas matemáticas para analizar dichos datos, pero a la hora de ofrecer unos resultados, sólo nos servirán aquellas comparaciones cualitativas, puesto que como se ha venido diciendo, el tiempo concreto que tardará en realizar las operaciones la aplicación depende en gran medida del hardware y sistema operativo sobre el que se está ejecutando.

Por otro lado recalcar que aunque se ha pretendido tener dos aplicaciones lo más parecidas posibles al inicio, hay algunos cambios mínimos que no han podido evitarse, que se comentan a continuación:

1-. Para empezar las medidas de tiempo. En el caso de **C++** se hace uso de la librería **`<sys/time.h>`**, que utiliza la siguiente estructura para representar la diferencia de tiempo:

```
double timeval_diff(struct timeval *a, struct timeval *b){
        return (double)(a->tv_sec + (double)a->tv_usec/1000000) -
                (double)(b->tv_sec + (double)b->tv_usec/1000000);
}
```

Por otro lado, en la aplicación **Java**, se utiliza el método **`nanoTime()`** de la clase **`System`**. De esta forma, aclarar que los distintos tipos de mecanismos utilizados para medir el tiempo pueden suponer unos errores mínimos en los tiempos tomados.

2-. *Payload*. En el caso de **C++**, para realizar el *payload* de modo que el número introducido por línea de comandos represente el número total de bytes a enviar, se ha utilizado la estructura **`Ice::Byte`**, perteneciente al paquete **Ice**, puesto que en **C++**, no hay un tipo "*byte*" explícito. La alternativa habría sido utilizar el tipo "*char*". Por otro lado, en **Java**, se utilizada el tipo "*byte*", que es la única posibilidad en el lenguaje, puesto que el tipo "*char*" de **Java** se corresponde con caracteres Unicode de 16 bits.

3-. En todas las aplicaciones se utiliza el *stream* hacia salida estándar para representar los datos que han ido obteniéndose. En este caso recalcar que en general, teniendo en cuenta la **JVM 1.6** de **Java**, el rendimiento a la hora de realizar operaciones de salida es similar aunque se mantiene algo superior en **C++**.

4-. Si se quieren realizar esas operaciones de IO hacia un fichero, como es lo lógico, puesto que conviene tener los datos recogidos en algún lugar, la cosa cambia, puesto que aquí el rendimiento de **C++** es bastante superior al que ofrece **Java**. Para no perjudicar la igualdad de ambas aplicaciones en este sentido, la solución tomada ha sido escribir en la salida estándar, y recoger mediante la sentencia de línea de comandos, la salida de la ejecución en un fichero de texto. Es decir:

En vez de $> java pub 1 cpu
$> java pub 1 cpu > time_cpu_java_1b.csv

De la forma anterior, el retardo de introducir la salida de las aplicaciones en el fichero de texto se delega al sistema operativo, y además se tratará del mismo retardo en ambos casos, evitando así esa posible diferencia que pudiera provocarse aquí.

La mayor diferencia existente a priori entre las dos primeras aplicaciones, es que en la interferencia de memoria, mientras que en **C++** se han podido asignar dinámicamente unos 90 megas aproximadamente, en la aplicación en **Java** sólo han podido asignarse 30, modificando además los parámetros de memoria de la **JVM** cada vez que debía ejecutarse la aplicación, y notando un evidente empeoramiento del rendimiento del ordenador donde se ejecutaba

instantáneamente. Esta gran diferencia se comentará de nuevo, cuando se comparen los resultados.

Por último mostrar el hecho de que se ha intentado en todo momento que las pruebas realizadas tuvieran el entorno lo más parecido posible, manteniendo los mismos programas abiertos y en general un consumo de recursos lo más parecido posible entre prueba y prueba. Además, el servicio **IceStorm** se ha ido reiniciando siempre entre prueba y prueba, para liberar recursos perdidos en la parte servidora, y del mismo modo para mantener un consumo de recursos lo más constante posible. Asimismo, para la prueba que mide el RTT, ésta se realizó en un ambiente completamente libre de carga en la red, siendo la única actividad existente en ese momento la que generaba la propia prueba.

4.3.2. Contraste y Comparativa de resultados

En la presente sección se presentan finalmente los resultados de las pruebas que se han realizado. Se compararán con gráficas, y se comentará sus aspectos relevantes para la comprensión del rendimiento de **IceStorm** en condiciones de sobrecarga.

De entre las mediciones existentes se observan tres grupos importantes, como se ha mencionado anteriormente. En las tres se realiza una batería de pruebas para las siguientes cargas: 1, 4, 16, 64, 256, 1024, 4096 bytes. A uno de los grupos se le ha asignado el nombre **Send Time**. Las pruebas de este grupo miden el tiempo en realizar un envío desde el "*publisher*". Es decir el tiempo será la suma de sacar el paquete por la tarjeta de red, más el "*marshalling*" de los datos. El segundo grupo recibe el nombre de **Total Time**, y las pruebas en este caso consisten en medir el tiempo correspondiente a la realización de 10000 envíos. El último grupo recibe el nombre de **RTT** y **RTT Localhost**. Estos dos hacen uso del servicio entero de **IceStorm**, calculando el RTT (*Round Trip Time*), tanto en un entorno distribuido, como en la máquina local *("localhost")*.

Los datos que se ofrecen se han calculado de la siguiente forma:

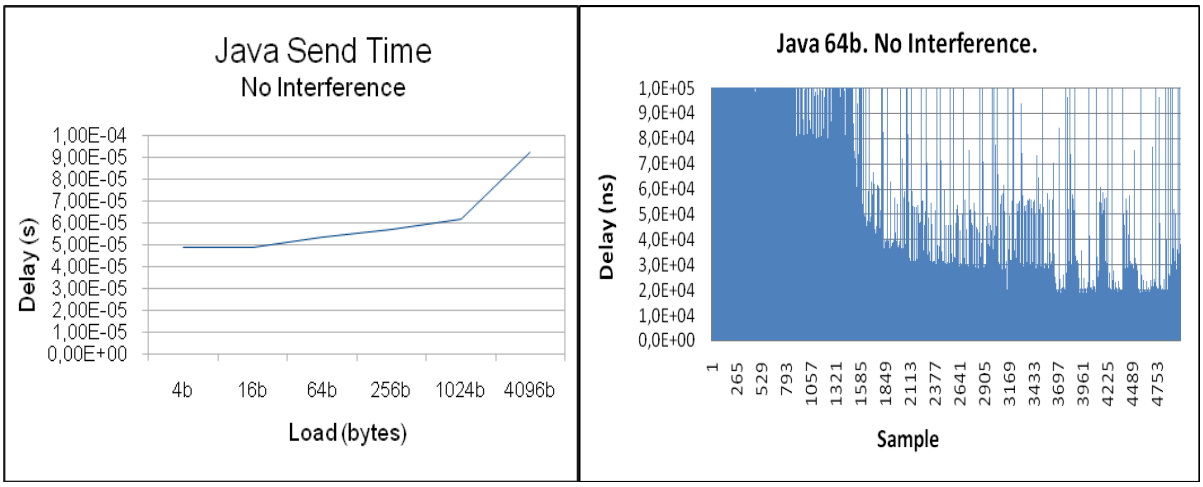
Send Time: Se realiza para cada *payload* y para cada tipo de interferencia, 10000 envíos. Por ejemplo, se configura la aplicación con los parámetros: *payload=1, Interference=cpu*, y tras esto se realizan 10000 envíos, lo que genera 10000 valores de tiempo, que se promedian para terminar con el valor ofrecido.

Total Time: Del mismo modo anterior se realiza para un *payload* y una interferencia concreta el promedio entre 20 muestras de tiempo. Cada muestra de tiempo corresponde en este caso a la realización de 10000 envíos.

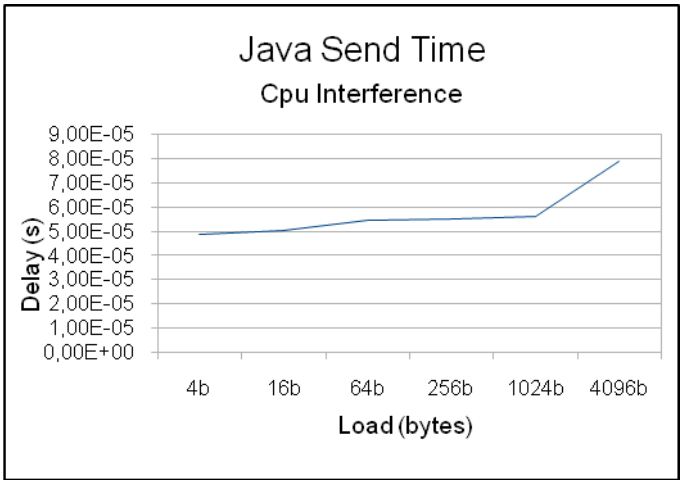
RTT y RTT localhost: Este caso es equivalente al de Send Time. Se toman para unos valores de *payload* e interferencia determinados 10000 medidas de tiempo, y se promedian éstos para ofrecer el valor.

A continuación se muestran las gráficas pertenecientes al Send Time para **Java**:

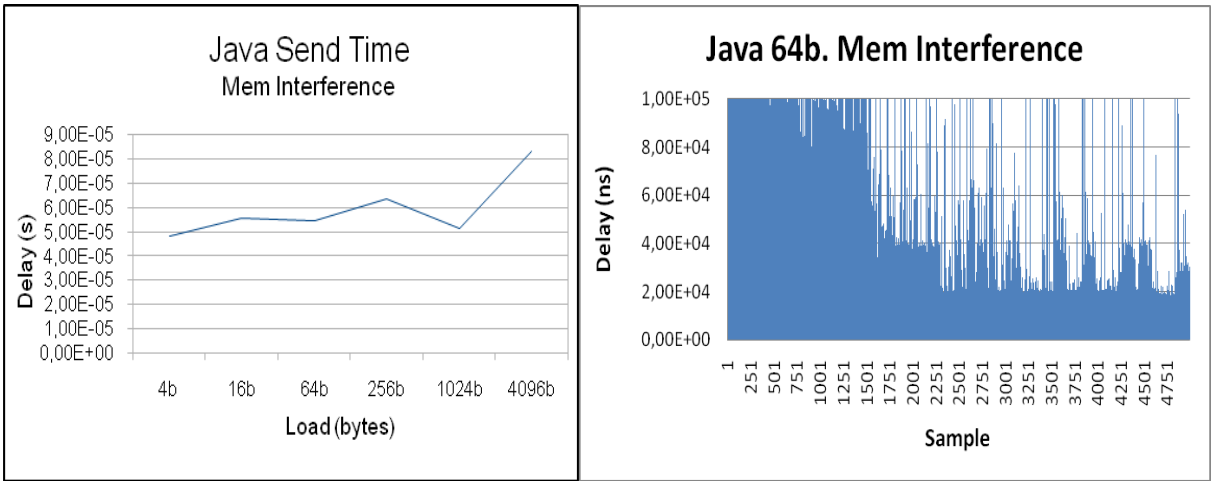
Las gráficas discretas que se presentan a continuación (izquierda) muestran en el eje X la carga que se ha utilizado ("payload"), y en el eje Y el valor medio de tiempo que ha llevado la operación para cada carga ("delay"). Como se ha comentado anteriormente, ese valor medio es resultado de promediar 10000 muestras. En los casos donde aparecen gráficas en la parte derecha, se presentan los tiempos correspondientes a las primeras 5000 muestras de tiempo obtenidas para un payload de 64b.



En este caso no se ha introducido ninguna interferencia, es el funcionamiento libre del "*publisher*" enviando datos. Se observa como la gráfica sigue una tendencia creciente conforme se aumenta el *payload*, lo cual es totalmente esperable. Ademas las operaciones se mantienen en unos rangos de tiempo similares hasta cargas de 1024 bytes, pero cuando se realiza la carga de 4096b, los valores se disparan. Esto último es lo primero reseñable que se encuentra durante la realizacion de las pruebas, puesto que con un *payload* de 4096b, los malos resultados son una constante. En la gráfica de la derecha se muestran las 5000 primeras muestras de tiempo para un payload de 64b, y puede apreciarse en dicha gráfica que la variacion no es uniforme, lo que supone una mala noticia, puesto que lo deseable es que el comportamiento sea siempre lo mas uniforme posible.



En esta gráfica vemos el comportamiento introduciendo una interferencia de cpu. El resultado es muy similar, no se aprecia una variación significativa para ningun valor de *payload*, lo cual es lo deseable.



En el caso de la interferencia de memoria, se observa que la gráfica se perturba de un modo visible, se aprecia claramente como los tiempos aumentan ligeramente, lo cual no es tan significativo como la perturbación que se aprecia, que desde luego, aunque no sea en gran medida, supone un punto negativo. Añadir ademas lo observado en la grafica de la derecha, que presenta los valores de las mediciones individuales para el conjunto de las 5000 primeras muestras, donde apreciamos que el comportamiento no es uniforme, sino que existe una gran variación desde los primeros tiempos a los ultimos.

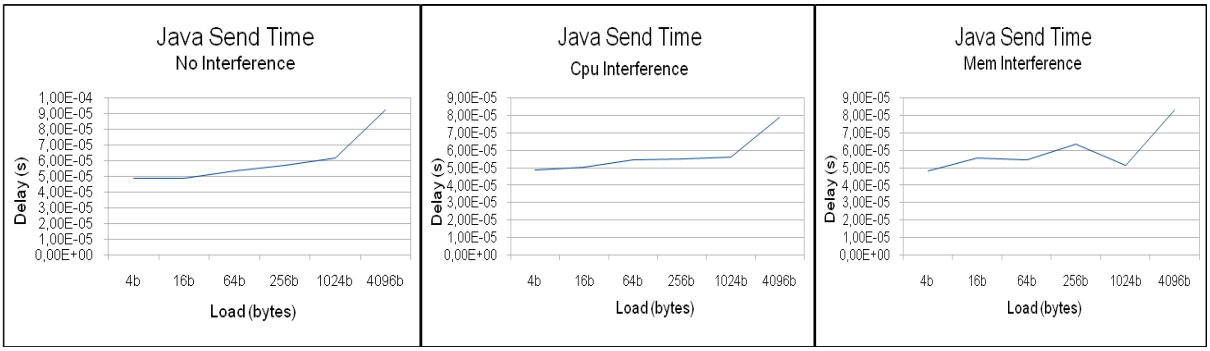
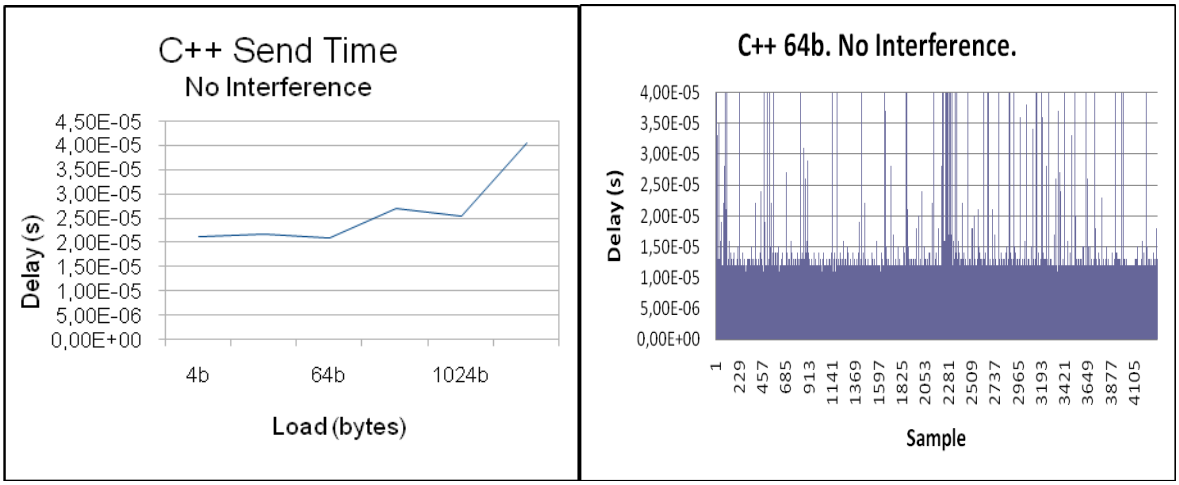
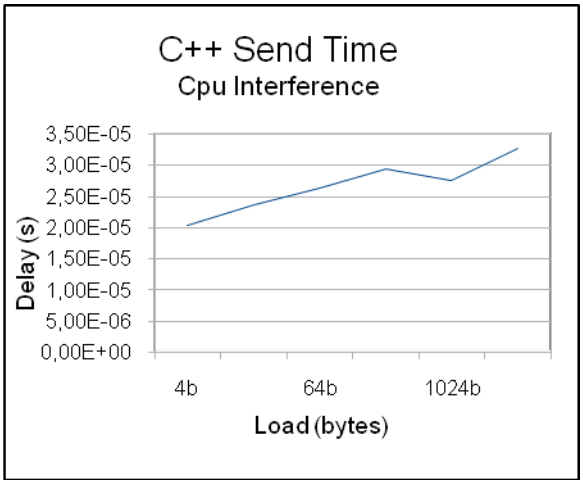


Figura 28: Send Time. Java.

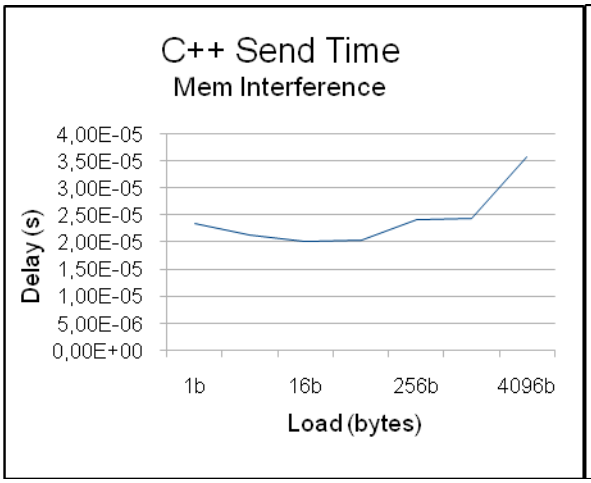
Tras ver los resultados para **Java**, pasamos a ver los correspondientes para **C++**:

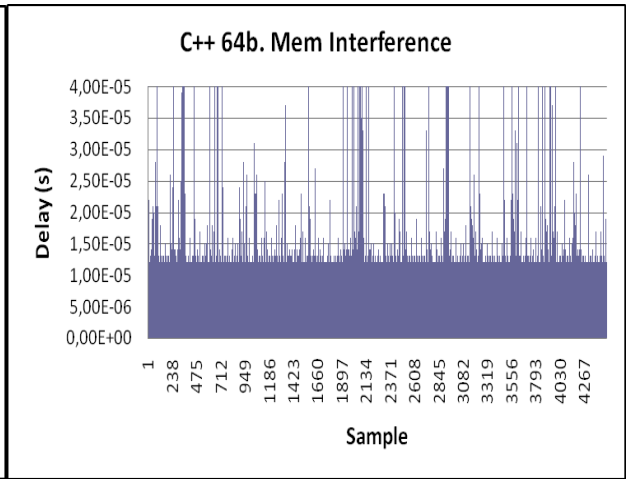


Se muestra primeramente el caso sin interferencia de ningun tipo. Se observa como al igual que en el caso de Java, la gráfica crece conforme aumenta el payload, hasta llegar al valor de 4096b donde se dispara el tiempo. Además es destacable fijarse en los valores de tiempo del eje Y, donde puede apreciarse como los tiempos que se manejan en este caso son aproximadamente la mitad de los conseguidos en las pruebas con Java. En la grafica de la derecha se aprecia como en el caso de C++, los tiempos de las muestras son mucho mas homogéneos conforme estas se suceden, al contrario que en Java, lo que supone una buena noticia para el rendimiento de Ice/IceStorm con C++.



Con la interferencia de cpu, se observa como la gráfica se ve más perturbada que en el caso de Java, donde permanecía aproximadamente constante. Esta mayor perturbación puede deberse a que al estar tratando en este caso con tiempos la mitad de grandes que en el caso de Java, la perturbación que genere la interferencia de cpu, por pequeña que sea, afecte en mayor medida al caso de C++.





Finalmente el caso de la interferencia de memoria. Vuelve a ser palpable el hecho de que los tiempos que se manejan son la mitad aproximadamente de los conseguidos con Java. Y se aprecia en la gráfica como la aplicación se ve mas perturbada por la interferencia de memoria que por la de cpu. Ademas en este caso, al contrario que en el caso de Java, si se observa la grafica de la derecha se aprecia como los tiempos individuales de las muestras se mantienen mucho mas constantes que en el caso de Java, siendo este un comportamiento mucho mas deseable.

Tras observar los resultados pertenecientes al Send Time, podemos indicar que la aplicación C++ viene siendo el doble de rapida que la de Java, y que la interferencia de memoria es la que afecta en mayor medida al rendimiento de IceStorm. Ademas, para un payload de 4096b, la aplicación, independientemente del lenguaje rinde mal. No obstante conclusiones más especificadas se encuentran en la ultima seccion de este capitulo.

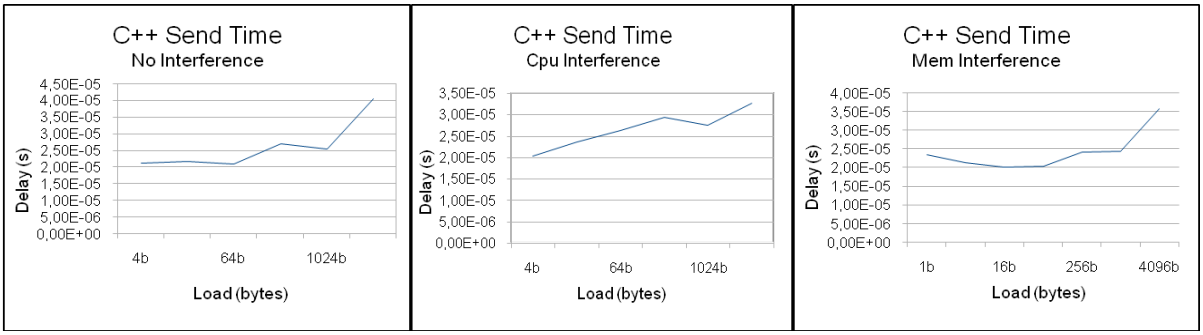
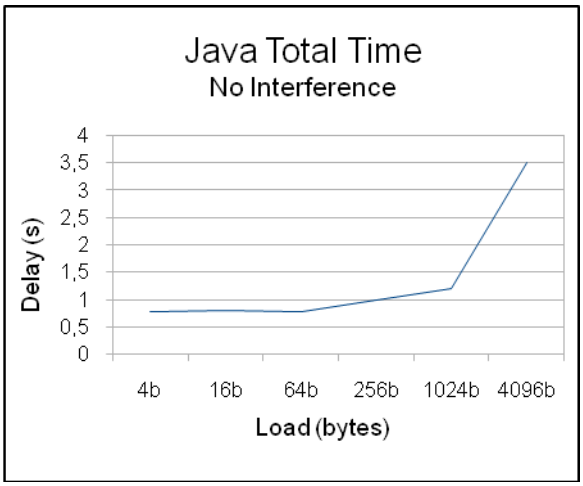
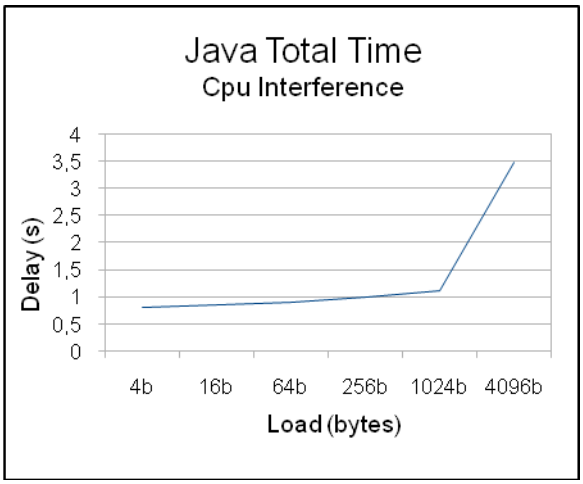


Figura 29: Send Time. C++

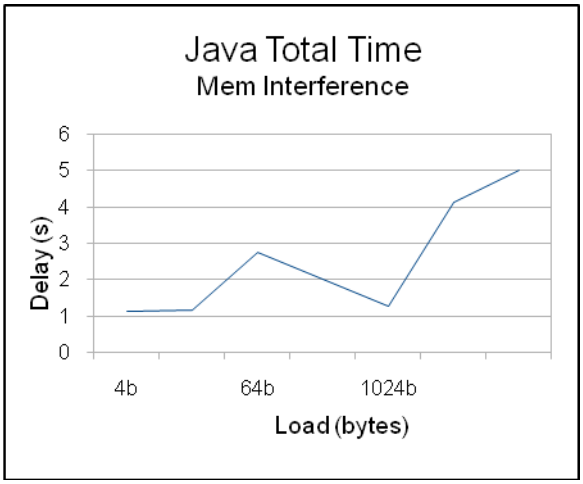
A continuacion se pasa a mostrar los resultados correspondientes al Total Time, para el lenguaje Java.



La gráfica anterior, que muestra los tiempos que consigue la aplicación Java para la prueba Total Time, nos ofrece unos resultados crecientes conforme aumenta el payload. Hasta llegar al valor de 4096b, donde vemos que el comportamiento es realmente malo.



En este caso se confirma lo que se comento en el caso de Send Time, la interferencia de cpu afecta de una forma realmente baja a la aplicación Java, ya que como se puede observar la grafica sigue una forma practicamente identica a la anterior, donde no habia interferencia alguna.



Otra situacion se observa en el caso de la interferencia de memoria, ya que como se aprecia en la figura, la gráfica presenta un comportamiento muy discontinuo, confirmando que la interferencia de memoria afecta en gran medida al servicio.

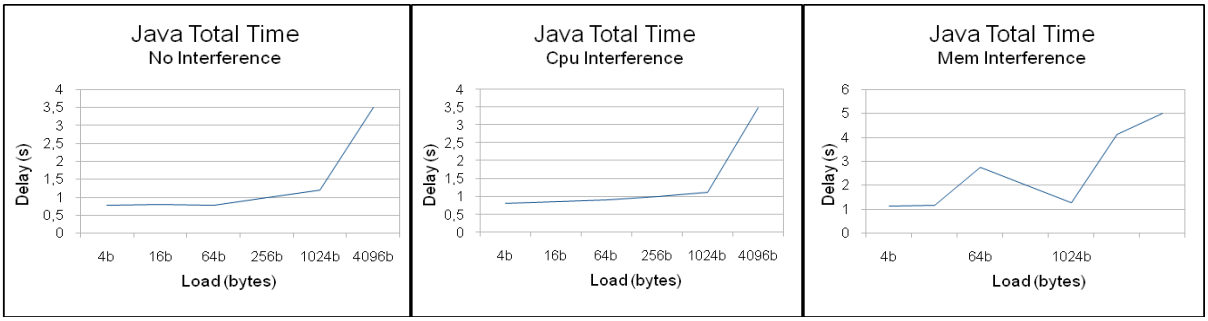
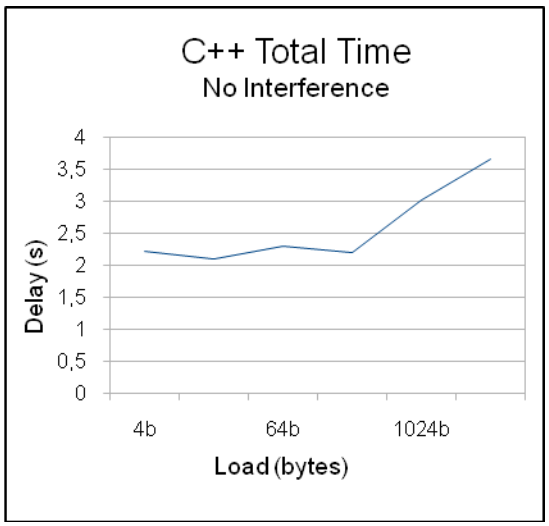
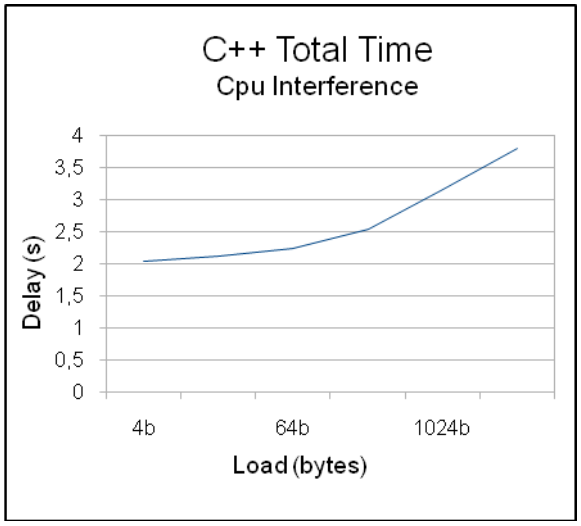


Figura 30: Total Time. Java

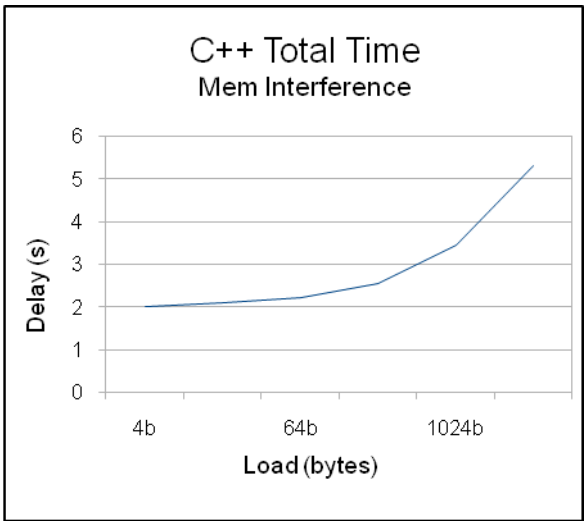
A continuacion se muestran los tiempos Total Time para la aplicación C++:



En esta primera gráfica, en la que no existe interferencia, vemos como el comportamiento es el que viene siendo habitual, creciente conforme aumenta el payload, y con un pico insalvable para un valor de éste de 4096b. Pero aparte de esta observacion se encuentra una que resulta cuanto menos curiosa, y es que los tiempos que se manejan aquí, son el doble que los que maneja la aplicación Java, justo al contrario de lo que sucedia para Send Time. La explicacion a este hecho se tratara de ofrecer en la seccion siguiente de conclusiones, pero se adelanta que tiene que ver con la JVM y las optimizaciones que realiza esta, asi como el recolector de basura existente en la plataforma de Java.



Con interferencia de cpu, nada adicional que destacar, sigue confirmandose que ésta interferencia afecta en una baja medida al rendimiento de IceStorm, tanto en la aplicación Java como en la C++, que es la que se observa arriba. Los tiempos como pueden apreciarse se mantienen en valores cercanos al doble que los correspondientes de la aplicación Java.



En este caso, con interferencia de memoria, apreciamos como al contrario que la aplicación en Java, dicha interferencia afecta en mucha menor medida a la aplicación C++, si bien es cierto que es la que consigue que se den peores resultados para payloads altos (a partir de 1024b). Este hecho nos sirve para pensar que C++ trata mejor los recursos de memoria que la de Java, lo cual por otro lado es obvio, como se comentara en la siguiente seccion, y tambien puede observarse que quiza los respectivos *mappings* de los dos lenguajes tengan algo que ver.

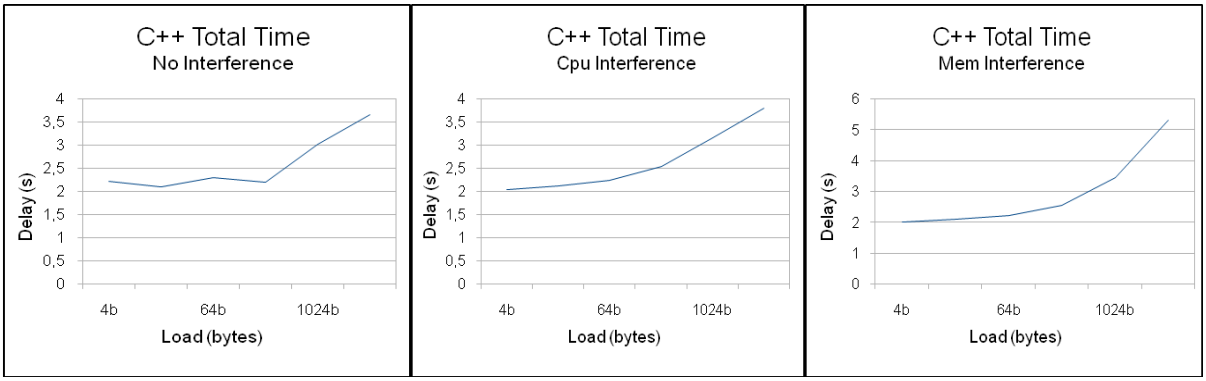
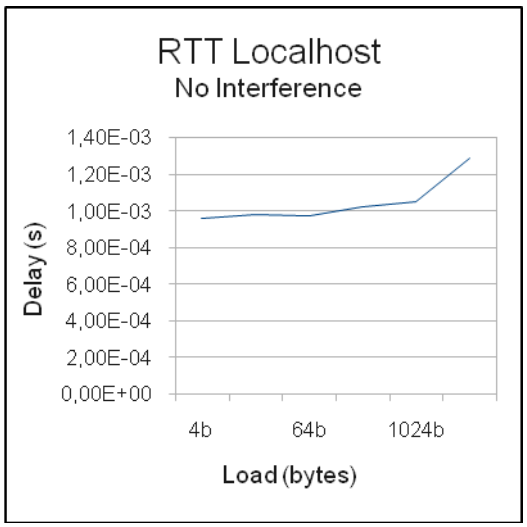


Figura 31: Total Time. C++

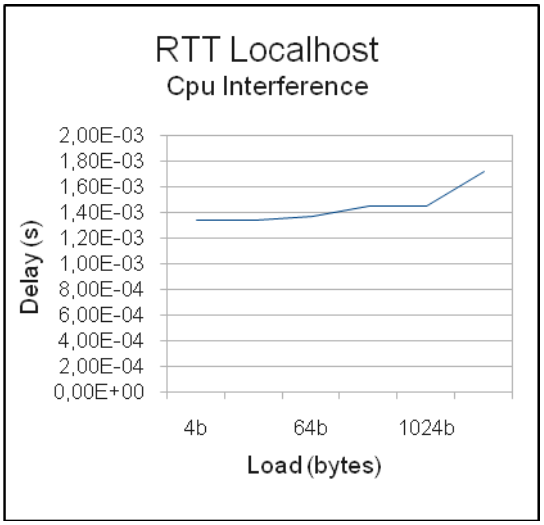
Tras observar los resultados pertenecientes a Total Time, puede concluirse que la aplicación C++ trata mucho mejor la interferencia de memoria que la aplicación en Java, aun siendo dicha interferencia distinta en ambos casos, como se comento en la seccion anterior (4.3.1). Ademas tambien ha podido observarse que los tiempos, de forma aparentemente inexplicable son el doble de largos en la aplicación C++ que en la aplicación Java, justamente al contrario de lo que podía observarse en Send Time.

A continuación se muestran las gráficas correspondientes a la prueba RTT Localhost.

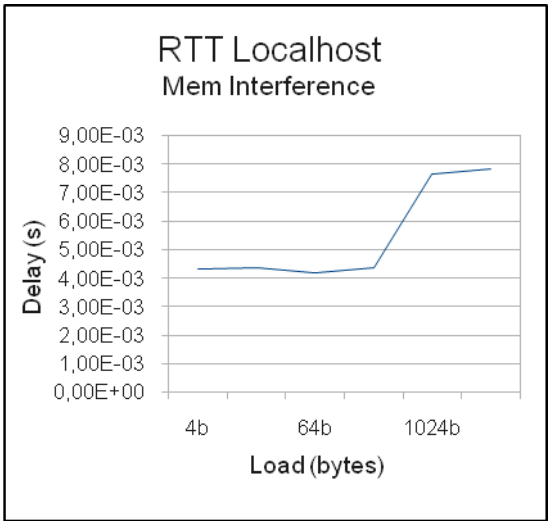


La grafica anterior muestra los tiempos RTT que da la aplicación cuando se ejecuta en localhost. Como se aprecia los tiempos son dos ordenes de magnitud

mas grandes que en el caso de Send Time. Es logico, puesto que en este caso, tenemos el Send Time del publisher en cliente, el Send Time del publisher en el servidor, y ésto sumado al tiempo que tardan en pasarse los datos por memoria compartida. Ademas se ve como la grafica permanece creciente, pero de forma muy ligera, hasta llegar al valor de 4096b de payload, donde se aprecia que el tiempo se dispara, como en los casos anteriores.



Con la interferencia de cpu en juego, el rendimiento empeora bastante, puesto que si se presta atención a los tiempos se ve un incremento importante. No obstante, es el resultado más lógico esperable, puesto que hay que recordar que en le misma máquina, además de la interferencia de cpu, están corriendo dos publicadores y dos suscriptores, además del servicio **IceBox** que gestiona otro servicio que también está corriendo, el propio **IceStorm**. Por tanto, la sobrecarga del sistema es mucho mayor, y de ahí el empeoramiento en los tiempos que se observa. No obstante no se aprecian picos, lo cual supone una buena noticia para el rendimiento de **IceStorm**, puesto que el comportamiento sigue siendo estable y más o menos uniforme.



Si el rendimiento con la interferencia de cpu se veia bastante afectado, la interferencia de memoria hace estragos en los resultados, ofreciendo tiempos, como se muestra en la grafica, cuatro veces superiores a los obtenidos sin interferencia. Este mal resultado es achacable al mismo motivo que para la memoria de interferencia, la gran cantidad de servicios que corren la maquina hacen que la importancia de la interferencia de memoria se vea aumentada. Ademas, como se puede concluir a raiz de las pruebas anteriores, la interferencia de memoria es la que mas afecta el rendimiento de IceStorm.

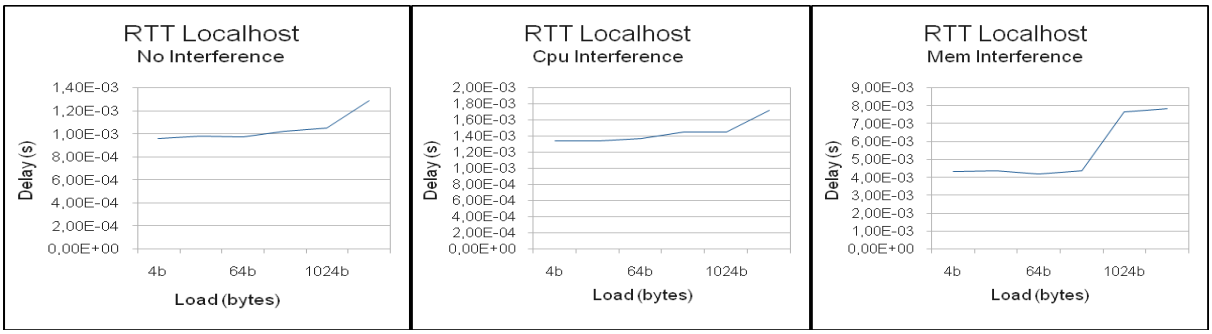
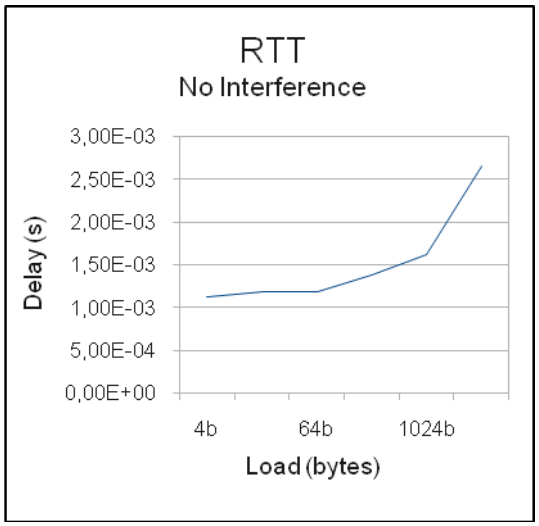
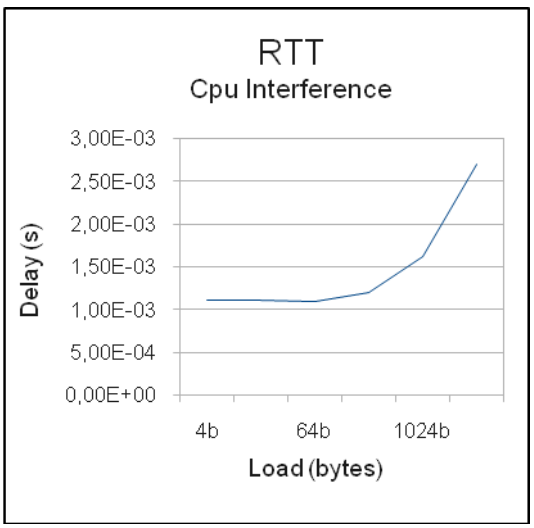


Figura 32: RTT Localhost.

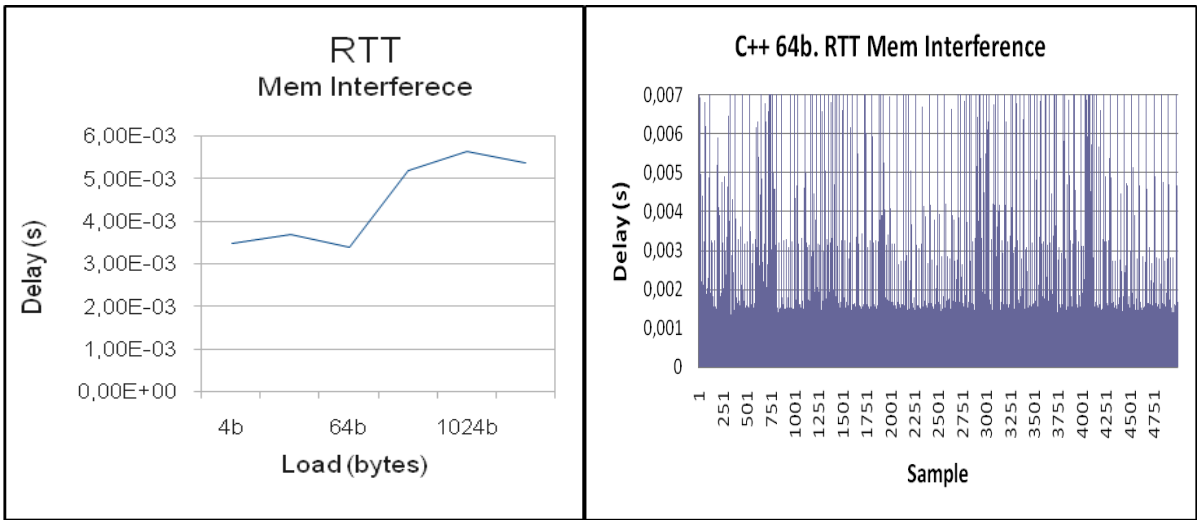
Tras haber presentado los resultados de RTT en localhost, a continuación se muestran los correspondientes al entorno distribuido, con dos nodos, y una red Ethernet real como unión.



En la anterior gráfica, que representa los tiempos en el caso de no existir ninguna interferencia, observamos como el comportamiento es estable, creciendo a medida que crece el "*payload*". En cuanto a los tiempos podemos apreciar que son mayores que en el caso de la prueba en localhost. Esto es logico, puesto que estamos en un contexto distribuido real, y la red conlleva un cierto retraso adicional. Especialmente el aumento de retardo introducido por la red se aprecia conforme el *payload* aumenta, de ahí que el comportamiento de esta grafica sea mas creciente que el de la propia en localhost.



En el caso de introducir la interferencia de cpu, podemos observar que el comportamiento es bastante parecido al propio sin interferencia. Como se ha venido comentando es debido a que la interferencia de cpu parece afectar de un modo muy leve el rendimiento, además en este caso, los distintos servicios que conforman la comunicación están distribuidos, por eso no vemos los grandes tiempos que se apreciaban en la aplicación propia en localhost. El comportamiento del servicio completo **IceStorm** en una comunicación completa, como es el caso, es positivo, puesto que permanece prácticamente inalterable, y se mantiene con consistencia ante la interferencia de cpu.



Finalmente la grafica correspondiente a la interferencia de memoria, donde la cosa cambia, y donde pueden apreciarse los picos que se generan, un comportamiento nada estable, y con tiempos que crecen hasta tres veces por encima de los propios para el caso sin interferencia y con interferencia de cpu. Ademas en este caso hay algunos otros aspectos a tener en cuenta que hacian del resultado de las pruebas incluso peores, pero se detallarán en las conclusiones. En la grafica de la derecha se muestra nuevamente la distribucion de tiempos para las 5000 primeras muestras correspondientes a un payload de 64b, donde se aprecia que existen una gran cantidad de picos, y donde se ve como el comportamiento no es demasiado estable. Por otro lado, cabe esperar que en el caso de haber utilizado Java para esta prueba, los resultados habrían sido desastrosos, ya que como se ha visto en ejemplos anteriores, la distribucion de los tiempos asociados a las 5000 primeras muestras en el caso de Java era muy inestable.

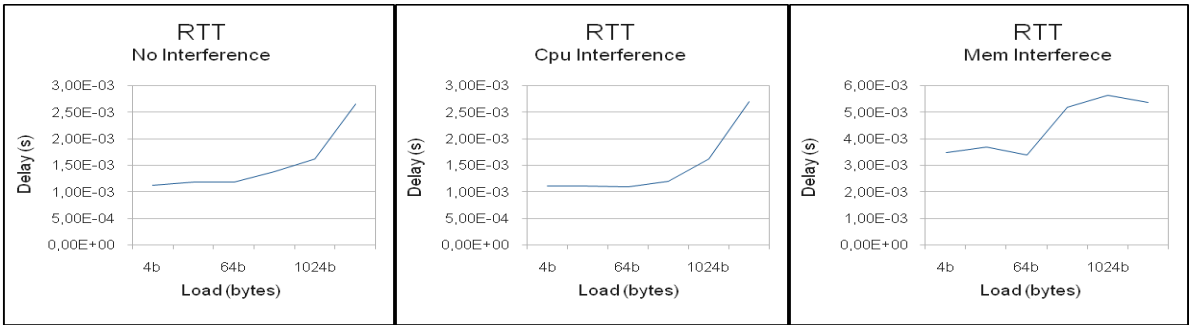


Figura 33: RTT

4.4. Conclusiones

Esta sección tiene como objetivo, realizar un pequeño resumen de lo visto anteriormente en el apartado de contraste y comparativa de resultados, con la intención de aunar en un espacio aproximadamente corto lo que ha dado de sí el estudio de **IceStorm**, presentando varias conclusiones a raíz, precisamente, de los resultados anteriores.

A rasgos generales pueden extraerse varias conclusiones objetivas, a partir de las dos primeras pruebas, **Send Time** y **Total Time**:

- El "*publisher*" en **C++** realiza en general un mucho mejor manejo de los recursos de memoria que el implementado en **Java**. Esta conclusión por otro lado tiene varias causas que no se pueden escapar, como son el hecho de la diferencia de paradigmas entre ambos lenguajes. Mientras uno es compilado (**C++**), y por tanto utiliza mejor los recursos del sistema, el otro es interpretado (**Java**), lo cual ya supone de antemano la utilización de un intérprete para tal código, como es la Java Virtual Machine, que por sí ya hace uso de unos recursos determinados.

- Cuando se trata del tiempo *marshall*, el correspondiente al envío de una carga por la red, se observa que los tiempos que consigue el "*publisher*" en **C++** son aproximadamente la mitad de los que consigue la implementación en **Java**.

- Por último, en el tiempo total, el tiempo en realizar 10000 envíos, la conclusión es opuesta a la anterior, puesto que en este caso es la aplicación en **Java** la que realiza dichos envíos en aproximadamente la mitad de tiempo que el Publisher en **C++**.

Por supuesto, cobra una importancia grande tratar de explicar los dos últimos puntos, que parecen contradecirse, pero antes conviene ofrecer más interpretaciones que no son quizá tan obvias pero que están ahí.

Para empezar, es necesario conocer un dato que no aparece en las gráficas ni en el resto de memoria, pues es algo que sólo es apreciable durante la experimentación, y por tanto difícil de documentar. Cuando se realizan las pruebas con ambos "*publishers*", se obtienen datos objetivos en cuanto a lo que han tardado en realizarse las operaciones que sean, y sobre un contexto igual para ambos "*publishers*". Bajo estas consideraciones, al observar las gráficas anteriores, especialmente en los casos de interferencia de memoria, se ven las diferencias entre las implementaciones en **C++** y **Java** que se han ofrecido más arriba, lo que no se aprecia es el rendimiento general de la máquina, que en el caso de la implementación en **Java** se hace por momentos intratable. Durante las pruebas con el resto de interferencias o con **C++**, apenas se nota que la máquina está realizando operaciones costosas en recursos, notando de este modo que la respuesta de los dispositivos de entrada, como son ratón y teclado se mantiene como siempre. En cambio, con la prueba de **Java** con interferencia de memoria la máquina queda mermada.

El caso anterior, se ha repetido en diferentes ocasiones variando algunos parámetros y siempre se ha observado el mismo efecto. A esto hay que sumar el hecho de que la interferencia de memoria para la implementación en **Java** es a priori menos agresiva que la equivalente para **C++**, puesto que de otro modo la máquina virtual ni si quiera aceptaba tal ejecución, aun indicando por línea de comandos que se quiere reservar una cantidad mayor de memoria que la de por defecto para la JVM.

Volviendo al tema anterior, insistir en que la implementación en **Java** tiene unos problemas mucho mayores con interferencia de memoria aun teniendo una interferencia mucho más suave que la propia de **C++**, lo cual debe tenerse también muy en cuenta a la hora de realizar interpretaciones de las respectivas gráficas.

Otro punto muy interesante es el que se ha dejado hasta ahora. ¿Por qué la implementación en **C++** es el doble de rápida en Marshall y la mitad de rápida en el tiempo Total?

La respuesta no es fácil, y tampoco se cuenta con suficientes datos para apoyarla, debido a la dificultad y tiempo que supondría incluir estos, tratándose de

un sistema operativo de propósito general. Se ofrece por tanto a continuación un acercamiento.

Se ha venido explicando a lo largo de este capítulo que lo importante a la hora de evaluar un middleware, es que éste sea apto para la aplicación donde vaya a utilizarse, y lo es normalmente cuando ofrece un rendimiento estable. Con esto no se habla de velocidad en ningún momento, sino de estabilidad. Se tiene la implementación en **Java**, que efectivamente realiza los 10000 envíos en un tiempo bastante menor que el que necesita el "*publisher*" **C++**. Pero algo que se escapa a los datos es que en el caso del "*publisher*" **Java** estos no son uniformes durante las 10000 muestras, y esto puede apreciarse en las graficas que se han incluido a la derecha de Java Send Time (No Interference y Mem Interference).

Cuando se realiza la prueba con el "*publisher*" **Java**, al principio se toma un largo tiempo en enviar el primer dato, tras lo cual la velocidad aumenta drásticamente, disminuyendo algunos órdenes de magnitud. Como ejemplo, en el tiempo total, donde se habla de apenas 1 segundo para realizar los 10000 envíos, (se habla de la media de las 20 veces que se realiza el experimento) la primera vez se han encontrado casos en los que el tiempo hasta realizar el primer envío superaba los 10 segundos. Estos casos se han eliminado, tratándose como muestras atípicas, puesto que en algunas ocasiones se convertían en un verdadero problema a la hora de ofrecer una visión del funcionamiento de **IceStorm**.

Además, una vez se ha pasado ese primer escollo, los tiempos que se generan desde la segunda muestra hasta la última van variando, disminuyendo conforme pasan muestras, lo cual tampoco suele ser un comportamiento deseado para casi ningún tipo de aplicación. Es un comportamiento inestable que también puede apreciarse observando las gráficas incluidas a la derecha de Java Send Time, sin interferencia y con interferencia de memoria.

Este comportamiento viene explicado por algunas optimizaciones internas que realiza la JVM en el transcurso de su ejecución y porque al inicio necesita preparar los recursos que va a utilizar. En el caso de **C++**, la ventaja que se encuentra es que aunque no se optimice en ejecución, ofrece un rendimiento muchísimo más estable que el ofrecido por **Java**. En cuanto a Java y su JVM, no podemos olvidar la función que realiza el recolector de basura, que no puede controlarse desde la aplicación y que puede actuar en cualquier momento.

Con estas consideraciones no se pretende desvirtuar la potencia y versatilidad de **Java**, simplemente aclarar que cuando se habla de un lenguaje interpretado, como es el caso, la estabilidad depende de más factores que en el caso de un lenguaje compilado, por lo que dependiendo de la aplicación, deberá utilizarse un lenguaje u otro. Por supuesto, si se cuenta con equipos potentes, y se ha calculado bien el consumo en recursos, **Java** puede ser una buena elección.

Por último, en cuanto a los lenguajes de programación utilizados, se comentó que la diferencia en los tiempos a la hora de realizar el *marshall* podía tener algo que ver con la diferencia de los *mappings* **Slice**. Como se aprecia al leer la documentación de **Slice**, la implementación que realizan en **C++** es la que cobra más importancia, tratando particularmente capítulos completos al uso de hilos en **C++** y a la optimización de las aplicaciones en **C++**. Además, **Ice**-**E** la especificación que hace ZeroC para adaptar **Ice** a dispositivos limitados utiliza como lenguaje **C++**. Esto no muestra otra cosa más que la importancia relativa que ofrecen en ZeroC por este lenguaje en detrimento de otros, pero también es cierto que comprobando las alternativas existentes, el único lenguaje que podría compararse en potencia, **Java**, sufre de otros problemas, que se reducen a los comentados más arriba. Por todos estos motivos, es claro que efectivamente puede que el *mapping* a **C++** sea mucho más óptimo que el propio a **Java**, unido a que **C++** realiza un mejor manejo de memoria, lo que tratándose de operaciones de serializado es altamente importante.

Teniendo en cuenta las pruebas **RTT** y **RTT Localhost**, se pueden llegar a las siguientes conclusiones:

- Es claro que en condiciones sin interferencia o con interferencia de cpu, **IceStorm** se comporta muy bien, manteniendo en todo momento un rendimiento uniforme para los distintos *payloads* que se prueban. Además atendiendo al **RTT Localhost**, en el que como se vio los tiempos aumentan con respecto a la variante sin interferencia, (lo cual se explicó debido al aumento de concurrencia que existía en el mismo nodo), podemos concluir basándonos en el comportamiento creciente que sigue dicha grafica, en que aunque el tiempo aumente, **IceStorm** es robusto además de rápido ante interferencias de cpu.

- Por otro lado, y nuevamente para confirmar las dificultades ante interferencias de memoria del middleware, se observa como ante éstas, el comportamiento decae en el servicio en general, encontrándose tiempos muy superiores al resto de situaciones de interferencia.

- En el caso de la medición de tiempos de RTT con interferencia de memoria, se observa como los resultados son malos en cuanto a tiempo. Pues bien, hay que unir a este hecho que la mitad de las muestras enviadas aproximadamente, carecían de la interferencia de memoria, puesto que el sistema operativo realiza algún tipo de gestión que recorta dicho consumo. (Las capturas que demuestran esto en el apéndice 5).

Para finalizar y sin salir del ámbito de middleware de propósito general, como se ha repetido tantas veces, **Ice** es estable y bastante robusto si se utiliza un lenguaje compilado, como pueda ser **C++**. La estabilidad del middleware sólo se ve afectada en casos donde el uso de memoria se lleva al límite de la máquina donde se encuentre el middleware.

Para su uso en sistemas embarcados, vista la robustez que mantiene con distintas cargas, y los pocos recursos que consume de la máquina, dejando en ésta una huella de memoria pequeña, se considera una buena opción.

En casos más generales para un middleware de este tipo, como deberían ser las redes de servidores que ofrecen servicios web a usuarios, donde los recursos no suelen plantear un problema, cualquier otro lenguaje para los que ofrece *mappings* puede ser muy apto, puesto que sin problemas de recursos, como se comenta, **Ice** es un middleware robusto.

Capítulo 5: Conclusiones y trabajos futuros.

5.1. Conclusiones

En este trabajo se comenzó sentando las bases actuales de algunas soluciones de middleware existentes en el mercado, tomando RTI **DDS** y ZeroC **Ice**, estudiándolas y realizando una comparación teórica entre ambas, tras la cual queda claro que se tratan de dos middleware enfocados a sectores del mercado con bastantes similitudes y algunas diferencias importantes. Por un lado **DDS** (concretamente la implementación de RTI que es de la que se disponía) es un estándar de publicación-suscripción del OMG, que se enfoca a aplicaciones críticas de tiempo real, y por otro lado ZeroC **Ice**, que es un middleware que surge como respuesta a **CORBA** para ser capaz de ofrecer una solución de menor complejidad que permite obtener programas mucho más ligeros para entornos embarcados; por lo tanto, el coste de desarrollo es significativamente menor que para CORBA y/o RTCorba.

Tras este primer capítulo, se toma **Ice** y se realiza un estudio del funcionamiento de dicha tecnología, centrando la atención concretamente en **IceStorm**, un servicio de **Ice**, que permite a éste implementar el paradigma de publicación-suscripción.

Para realizar dicho estudio del funcionamiento, se propone un demostrador de videovigilancia, en el que se adaptan una serie de requisitos para conseguir tocar las partes esenciales de **Ice**, como son el lenguaje **Slice**, que aporta facilidades de independencia de plataforma. Además, se despliega dicha aplicación en un sistema distribuido real, compuesto por dos ordenadores interconectados por un switch en una red Ethernet.

La función de dicha aplicación no es otra que suponer un buen punto de partida para realizar una implementación de una aplicación real, y tener una base que sirva para fines didácticos. Además de hacernos una idea de las características del proceso de desarrollo que supone la utilización de **Ice**.

Tras la explicación exhaustiva de la aplicación realizada, se encuentra un estudio del rendimiento de este middleware. Sin realizar comparaciones con

otros middleware que puedan encontrarse en el mercado, el estudio pretende aportar información básica para ayudar en el proceso de decisión en la elección de un middleware para entornos de red embarcados que requieren un cierto rendimiento. Para esto se disponen de tres aplicaciones de prueba, en **C++** y en **Java,** para realizar ciertas medidas básicas tanto en un ambiente distribuido como en la máquina local. Por tanto, se estudia el rendimiento desde el punto de vista del "*publisher*", y se mide el rendimiento de **IceStorm** de un modo más general. En las pruebas se varía la carga, *payload*, y la interferencia.

Después del estudio de rendimiento se concluye que **IceStorm** es un servicio robusto si se utiliza un lenguaje compilado y no hay demasiados problemas de recursos. Por otro lado, si la idea es variar alguna de las dos premisas anteriores, la estabilidad puede verse afectada ligeramente, y lo hace si la cantidad de memoria disponible es poca.

En general, al hablar de **Ice**/**IceStorm**, se habla de un middleware de propósito general, que hace un gran hincapié en servicios web, y que posee un muy buen rendimiento con **C++** como base.

Este trabajo se ha encuadrado dentro de los inicios de un proyecto europeo en el que los sistemas de videovigilancia son uno de los dominios de aplicación.

5.2. Trabajos Futuros

En la última sección de este trabajo se tratan las posibles continuaciones que pueden dársele a este.

ICE tiene unas características que le hacen particularmente interesante para ser utilizado en infraestructuras distribuidas con nodos embarcados, ofreciendo además una serie de lenguajes que cubren por completo las necesidades que pueden encontrarse en tales escenarios. Un posible trabajo futuro sería desplegar una aplicación orientada a servicios, con **Ice** como el middleware en la infraestructura, ya que sería un buen modo de ver otro de los ámbitos donde los middleware están tomando fuerza en los últimos años.

El trabajo más directo sería, posiblemente, ampliar la demo de videovigilancia, permitiendo decenas de cámaras (*publishers*) y decenas de servidores donde se recoja la información generada (*subscribers*). De esta forma, podría evaluarse el middleware con una aplicación más realista. Además, a modo de estudio adicional podría aumentarse progresivamente el número de cámaras y servidores, para ver cómo reacciona **Ice** ante el aumento de concurrencia.

Por otro lado, podría tomarse **Ice**-**E**, la especificación desarrollada para sistemas limitados, y probarla con una mayor gama de dispositivos embarcados, portándola previamente a algunos de los RTOS (Real-Time Operating System), para que encajara en éstos. Al respecto, existe una implementación de **Ice**, no oficial, portada a QNX, por la Universidad de Sidney, más concretamente por el "*Australian Center for Field Robotics*". Al parecer no es especialmente complicado realizar este porte, por eso se plantea como una posibilidad. En el caso de **Ice**-**E**, cuya implementación es incluso más reducida, podrían ganarse algunas ventajas, y tomando algunas otras consideraciones a la hora del desarrollo del middleware, podría derivarse una tecnología apta para un mayor número de dispositivos.

Portar **Ice**-**E** a sistemas operativos de tiempo real tendría diversas ventajas, entre las que se encontraría, por ejemplo un mayor control a la hora de manejar la concurrencia y a la hora de preparar aplicaciones para dispositivos más

limitados, puesto que se conseguiría un mayor control sobre los recursos del sistema. Por otro lado, sería el primer paso a realizar si el middleware **Ice** quiere escalar en otros sectores del mercado, mostrándose como una alternativa eficaz ante aplicaciones de redes de sensores, de sistemas domóticos, etc. Esto último sería útil, dada la buena estructura general que presenta el middleware, la cual facilita mucho su implantación en las aplicaciones para las cuales está diseñado, ya que trasladaría dicha funcionalidad y facilidad a aplicaciones que cobrarán una gran importancia en un muy corto plazo.

En general, y a parte de las evidentes mejoras que podrían realizarse sobre la aplicación demo, como serían el ajuste de ésta para conseguir una aplicación más realista, podría trabajarse en un estudio más exhaustivo del rendimiento de **Ice**, pero esta vez realizando una comparación con otros middleware actuales, como puedan ser **Java RMI**, o **WCF** de Microsoft, para ofrecer otra comparativa que pueda servir como guía a la hora de plantear la utilización de un middleware u otro.

5.3 Presupuesto

En la presente sección se presenta el presupuesto estimado para la realización de este proyecto, en el que se incluyen los costes tanto materiales, personales, como temporales.

Para la realización del proyecto es necesario disponer de un ingeniero a jornada completa.

Se estiman 15 días para documentación e instalación del middleware en una máquina. En este tiempo es importante que consiga un buen manejo básico del middleware en varias plataformas, para conseguir la mayor rapidez a la hora de emprender la tarea de desarrollo.

Tras los 15 primeros días, comienza la labor de diseño y de búsqueda de tecnologías para resolver los problemas que se enfrentan en el desarrollo de la aplicación. Son necesarios otros 12 dias para plantear un diseño coherente de la aplicación de videovigilancia y elegir las tecnologías que se utilizarán.

Se estiman en este momento unos 25 dias de desarrollo de la aplicación de videovigilancia, contando con control de errores y con otras comprobaciones.

Una vez desarrollada la aplicación de videovigilancia, se debe realizar el estudio. En este punto se calculan aproximadamente 5 dias para buscar información sobre *benchmarkings* a middleware y para plantear las pruebas que son necesarias realizar. A esto le siguen 12 dias de desarrollo de las aplicaciones de prueba y de la realización de las correspondientes pruebas.

Una vez realizadas las pruebas correspondientes, se calcula un tiempo de unos 15 dias para reflexionar sobre el trabajo realizado, establecer conclusiones, y preparar una memoria que muestre la experiencia con el middleware utilizado, asi como la descripción de la aplicación de videovigilancia desarrollada y la descripción detallada de las aplicaciones de prueba utilizadas, como de los resultados alcanzados con estas.

Costes:

Personales:

Suponiendo un sueldo del ingeniero de 40€/hora, el coste total del tiempo que se invierte en el proyecto es:

40€ x 8h = 320€/día.
320€ x 84dias = 26880€

En total por tanto: *26880€.*

Material:

Dos PC's, con un coste por unidad de 800€.

800€ x 2 = 1600€.

Dos monitores planos TFT:

300€ x 2 = 600€.

Un Switch de altas prestaciones, como el usado en realidad. TRENDNet TE100-S24 10/100 Mbps NWay Switch

350€ x 1 = 350€

Cable Ethernet:

30€

Alquiler de un estudio durante 4 meses:

900€ x 4 = 3600€.

Material fungible:

200€

Cuota de conexión de altas prestaciones a Internet durante 4 meses:

80€ x 4 = 320€

En total: *6700€*

Software:

En este caso todas las herramientas usadas para la realización del proyecto son gratuitas, tanto SO, como middleware, como drivers para la cámara, como herramientas de edición, etc.

Por tanto, en total, el **presupuesto del proyecto es de 33580€.**

Anexos

Introducción

Esta sección está destinada a ofrecer unos pasos sencillos para poder utilizar las tecnologías utilizadas durante el proyecto.

Primeramente se ofrecerán dos anexos en los cuales se mostrará cómo debe realizarse la instalación de la librería **V4L4J**, y del middleware **Ice**. Ambos anexos estarán enfocados a los pasos necesarios para tener estas dos tecnologías instaladas en sistemas basados en Debian.

Tras estos primeros anexos, se ofrecerá un anexo más en el que se mostrarán todas las labores de configuración que son necesarias además de lo comentado para que la aplicación de videovigilancia pueda funcionar. Entre estas configuraciones cabe destacar las asociadas al middleware **Ice**, puesto que se necesitara tener cierto manejo del servicio **IceBox** para poner a funcionar otros servicios, como son **IceStorm**, que como se ha visto, es ampliamente utilizado durante el desarrollo del proyecto.

Además, **Ice** basa la mayoría de sus configuraciones en ficheros de texto, en los cuales se representan una serie de líneas formadas por pares de atributo-valor. Pues se procurará mostrar aquellas más relevantes, explicando qué consideraciones es necesario tener en cuenta para llegar a cierta funcionalidad. Asimismo es importante mantener siempre un código lo más desacoplado posible de los parámetros de configuración que puedan variar, para evitar tener que recompilar todos los ficheros y ejecutar con la nueva configuración. **Ice** permite esto mediante estos ficheros de texto, y será otro de los puntos en los cuales se incidirá.

Se ofrece asimismo un anexo donde se muestran capturas de la aplicación en funcionamiento, donde se ve el aspecto de la interfaz gráfica, etc.

Finalmente se muestra el código de las aplicaciones de prueba, utilizadas para realizar la evaluación de **IceStorm**, como se indica en el capítulo 4 del documento.

Anexo 1: Instalación de la librería v4l4j

Video 4 Linux 4 Java es un wrapper en **Java** de los drivers **V4L** nativos, los cuales están escritos en C. **V4L4J** ofrece la misma funcionalidad que **V4L** pero para el lenguaje de programación **Java**.

En este anexo se va a explicar cómo instalar la librería para poder utilizarla en una aplicación, compilándola desde su código fuente para ampliar el abanico de plataformas para el cual pueda usarse. Es importante indicar que todos los comandos que se indican en el anexo son posibles en sistemas basados en Debian, si el sistema en cuestión no es Debian o basado en él, entonces será necesario adaptar estos comandos.

Antes de empezar con la instalación propia de la librería, hace falta cubrir una serie de dependencias que necesita ésta.

1-. Es necesario tener la última versión del JDK instalada en el ordenador, en estos momentos, la JDK 1.6.

```
$sudo apt-get install sun-java6-jdk
```

2-. También se necesita la librería de utilidades JPEG.

```
$sudo apt-get install libjpeg-dev
```

3-. Por último es necesario disponer de ant para las labores de compilación. Esto puede conseguirse de la siguiente forma:

```
$sudo apt-get install build-essential ant
```

Una vez se tienen todas las dependencias satisfechas, se muestra a continuación una serie de pasos que son los que se deben realizar para conseguir la instalación de la librería correctamente. Se repite que los pasos seguidos son para

sistemas basados en Debian, como es el caso de Ubuntu, la plataforma donde se ha probado la aplicación. Si no disponemos de esta plataforma, será necesario adaptar dichos comandos.

1-. Descargamos de la página del proyecto los ficheros fuente de la librería. En el caso de este proyecto se ha utilizado la versión 0.8.3, pero no debería suponer ningún problema usar una versión posterior a esta.

http://code.google.com/p/**V4L4J**/downloads/list

En esta página se busca el fichero correspondiente a los archivos fuente. En la columna “summary+labels”, debe aparecer algo como:

Stable <version> source archive.

Se descarga por tanto ese fichero y se guarda en alguna carpeta donde se tengan todos los permisos. Lo más común es en espacio de usuario.

2-. Una vez han sido descargados los ficheros fuente, se extraen en una carpeta, y nos movemos a la raíz de dicha carpeta desde un terminal.

3-. Dentro de la carpeta se encuentran varios ficheros, entre los cuales debe encontrarse un build.xml, que será el fichero raíz desde el cual ant extraerá toda la información que le sea necesaria.

4-. Una vez comprobado que se encuentra este fichero, debemos ejecutar el siguiente comando:

```
$sudo ant install
```

Este comando configurará todos los ficheros necesarios, los compilará y los instalará, es decir moverá los ficheros a sus lugares por defecto. Estos lugares por defecto son:

/usr/lib/jni -> Aquí se moverá la librería **libv4l4j.so**

/usr/share/java -> Aquí se moverá la librería **V4L4J.jar**

Si el anterior comando no ha dado errores, podemos ir a los anteriores directorios para comprobar que se encuentran dichos ficheros. Destacar en este punto que libv4l4j.so es la librería en C, y **V4L4J**.jar es la librería de **Java**.

Si todo ha resultado correcto ya está todo listo para ser utilizado. Podría ocurrir un error si no fuese capaz de mover los ficheros a los directorios. En este caso tras ejecutar el comando se observará en la carpeta raíz ambas librerías, pero no podremos utilizar aún **V4L4J** porque estas librerías no se encuentran en sus respectivos lugares. Para subsanar este problema no tiene más que moverse manualmente estas librerías a sus respectivos directorios.

Una vez todo este en su sitio, estamos listos para utilizar **V4L4J** en nuestra aplicación. Para esto, hay que indicar los siguientes parámetros a la hora de compilar y ejecutar la aplicación que hayamos desarrollado.

```
-Djava.library.path=/usr/lib/jni -cp /usr/share/java/V4L4J.jar
```

Simplemente estamos incluyendo indicando a la JVM donde debe hallar las librerías que le serán necesarias para compilar o para ejecutar la aplicación en cuestión.

En cualquier momento, podemos desinstalar la librería ejecutando la siguiente sentencia en línea de comandos desde la carpeta raíz donde encontraremos el fichero build.xml.

```
$sudo ant uninstall
```

Anexo 2: Instalación del middleware Ice 3.3.1

En este apartado se va a explicar la forma más directa posible para instalar el middleware en un sistema basado en Debian.

Lo primero relevante que hay que tener en cuenta es que en la versión 3.3.1 del middleware no hay soporte oficial para sistemas basados en Debian, solo para ciertas plataformas, que manejan paquetes RPM, entre los que se encuentran Suse, o Red Hat.

No obstante, **Ice** es software libre, y se facilita los ficheros fuente, de modo que pueden compilarse para casi cualquier plataforma, aunque eso sí, no es soportado de forma oficial.

Del mismo modo que en el apartado anterior, el primer paso que debe llevarse a cabo es instalar todos aquellos programas o librerías que son dependencias de la plataforma **Ice**.

Berkeley Database 4.6.21:

Esta aplicación es necesaria para todos aquellos servicios de **Ice** que requieran ciertas facilidades de persistencia. En nuestro caso **IceStorm** requiere esta aplicación, puesto que en su funcionamiento por defecto, se mantienen datos de forma persistente.

En los repositorios de Ubuntu puede encontrarse dicha aplicación. Debemos instalar tanto el fichero principal, como el fichero “utils”, así como las librerías de runtime de **C++** y de **Java** (en el caso de que quisiésemos utilizar el servicio en **Java**).

MCPP:

Se trata de un preprocesador de C. Durante la compilación de **Ice**, es necesario tenerlo para ciertas tareas que en la mayoría de casos permanecen transparentes para el usuario.

La forma más fácil de instalar esta dependencia es visitar la página donde se aloja el proyecto.

http://mcpp.sourceforge.net/download.html

Y descargar la versión 2.7.2, que es la necesaria para la versión utilizada del middleware. Tenemos la opción de descargarlo en varios formatos para varias plataformas, la más sencilla será descargar el paquete .deb, de modo que podamos instalarlo directamente en el sistema basado en Debian.

Tras instalar estas dos primeras aplicaciones, faltan aun otras dos más para que todo funcione correctamente. La opción más rápida es obtener éstas de la página de ZeroC **Ice**, ya que ofrecen un archivo comprimido con todas las dependencias que es susceptible de necesitar la plataforma. Por ello, vamos a la página:

http://www.zeroc.com/download.html

Aquí bajamos hasta la sección Source Distributions, la subseccion Third-Party Source Code, y descargamos el fichero correspondiente a los fuente para sistemas Linux.

Una vez descargado, se descomprime el fichero, y se entra en la carpeta para ver todas las aplicaciones que son dependencias. Nos interesan dos en concreto, bzip2 y expat. A continuación se muestran los pasos necesarios para instalar cada una de ellas.

BZIP2:

Tras descomprimir la carpeta donde se encuentra esta aplicación, se entra en raíz. Una vez aquí, desde un terminal se ejecutan los siguientes comandos:

```
$ make
$sudo make install
```

De esta sencilla forma se compilan e instalan los ficheros necesarios.

EXPAT:

Procediendo del mismo modo que con bzip2, extraemos el contenido de la carpeta expat y nos movemos a raíz. Seguidamente tecleamos desde un terminal:

```
$ ./configure
$ make buildlib
$ sudo make installlib
```

Una vez hecho esto ya se habrá terminado de instalar todas las dependencias necesarias, base para instalar el middleware **Ice**.

Instalación de Ice 3.3.1:

Como se comentó al inicio del anexo, no hay soporte oficial para **Ice** bajo plataformas basadas en Debian. No obstante, es posible instalarlo, y para ello hay varias alternativas. La alternativa más correcta sea quizá compilar los ficheros fuentes para nuestra plataforma en concreto, pero ésta es una tarea muy tediosa en el caso de **Ice**, y por ello se va a hacer uso de una alternativa más rápida y limpia igualmente.

La alternativa de la que se habla es la utilización del programa alien para

realizar transformaciones entre paquetes de distinto tipo. De este modo, lo que se hará será transformar aquellos paquetes .rpm que sean necesarios en paquetes .deb, aptos para nuestro sistema basado en Debian.

Para esto, lo primero necesario es instalar precisamente el programa alien, el cual se encuentra en los repositorios de Ubuntu, por tanto se puede realizar la instalación siguiendo los procedimientos normales del sistema en concreto.

Tras tener el programa alien instalado, es necesario acceder a los paquetes que será necesario instalar. Para esto descargamos nuevamente de la página de ZeroC el fichero **Ice**-3.3.1-sles10-i586-rpm.tar.gz, una vez extraído convenientemente, deben transformarse los siguientes paquetes:

- **Ice**-3.3.1-1.sles10.noarch.rpm
- **Ice**-**C++**-devel-3.3.1-1.sles10.i586.rpm
- **Ice**-libs-3.3.1-1.sles10.i586.rpm
- **Ice**-servers-3.3.1-1.sles10.i586.rpm
- **Ice**-utils-3.3.1-1.sles10.i586.rpm
- **Ice**-java-3.3.1-1.sles10.noarch.rpm
- **Ice**-java-devel-3.3.1-1.sles10.i586.rpm

Para transformar dichos paquetes en formato .deb, puede usarse alien desde línea de comandos de la siguiente forma.

```
$alien –k <nombre_paquete>
```

De este modo obtenemos el correspondiente .deb de cada paquete. Y una vez lo hayamos realizado para todos ellos solo hará falta instalar, que al ser paquetes .deb se convierte en una tarea sencilla. Solo hay que tomar en cuenta el

orden que aparece arriba, para realizar la instalación en ese mismo orden, de modo que no se incumplan dependencias durante el proceso de instalación.

Anexo 3: Configuración del servicio IceStorm

Este tercer anexo tiene como fin mostrar el funcionamiento de **Ice** en general, y en particular del servicio **IceStorm**, para lo cual se explicará el funcionamiento de ambos, para pasar después a la configuración específica que encontramos en la aplicación de videovigilancia.

Ice utiliza un mecanismo de configuración basado en ficheros de texto. De este modo en dichos ficheros podemos configurar parámetros tanto de la comunicación como de otros aspectos de **Ice** o de cualquiera de sus servicios. Este mecanismo permite mantener un código desacoplado y más limpio, ya que aquellas configuraciones susceptibles de ser modificadas se mantendrán fuera del código, en un fichero de texto, que podremos modificar sin ver la necesidad de recompilar nuestro programa.

Mantener los parámetros de configuración en ficheros de texto dota, por tanto, de una versatilidad extra a las aplicaciones. **Ice** provee un pequeño y sencillo API para acceder a las configuraciones de los ficheros y cargarlas en tiempo de ejecución, para evitar la necesidad de parar la aplicación al cambiar una configuración. Visto de este modo, parece lógico pensar que un mecanismo de estas características era completamente necesario para una tecnología de middleware que en principio se presupone que debe funcionar ininterrumpidamente en aquel ambiente para el cual esté destinada, independientemente de la aplicación concreta.

Los ficheros de configuración, se componen de una serie de pares atributo-valor, siendo el atributo la propiedad configurable en concreto, y el valor aquel que se haya asociado a dicha propiedad. Los atributos existentes están definidos perfectamente en el manual de referencia de **Ice**, así como los valores que es posible asociar a cada uno de ellos. De este modo podemos indicar por ejemplo que el tamaño máximo del mensaje será de 1024 bytes con la siguiente línea:

Ice.MessageSizeMax = 1024

La propiedad MessageSizeMax define el tamaño máximo de mensaje como podemos observar en el manual de **Ice**, por tanto del modo anterior decimos que tal tamaño será de 1024u.

Conocer el mecanismo de configuración de **Ice** es una tarea obligatoria para poder sacar un mejor rendimiento de la tecnología.

Visto por encima el mecanismo de configuración, es necesario indicar cómo accede el servicio **Ice** a dichos parámetros. Existen varias alternativas para este fin. Por defecto, **Ice** accederá al fichero que apunte la variable del sistema **ICE**_CONFIG, donde se presupone estará la configuración por defecto. De esta forma, podemos apuntar nuestro fichero de configuración en la variable **ICE**_CONFIG, y por defecto cuando arranquemos el servicio leerá las propiedades que figuren en dicho archivo. Otro modo, algo más versátil para algunos casos, es indicar por línea de comandos el fichero de configuración que debe utilizar la aplicación, de este modo podemos tener preparados varios con distintas configuraciones y utilizar alguna de ellas sin tener que modificar ni si quiera los ficheros de texto. Por último podemos indicar también por línea de comandos la propiedad en concreto que queremos modificar con su valor correspondiente, siendo esta última opción la de mayor prioridad, por lo que si **Ice** está cargando un fichero por defecto, en el que aparece un atributo A con un valor x, y por línea de comandos se indica que el atributo A tendrá valor y, impera este último valor, configurándose el servicio, por tanto, con el valor y en la propiedad A.

Dicho esto vamos a pasar a observar los ficheros de configuración utilizados para la aplicación de videovigilancia, explicando algunas de las propiedades importantes que aparecen en éstos, para lo cual es importante saber antes, que **Ice** gestiona todos sus servicios a través de un servicio en sí, cuyo nombres es IceBox, y cuya funcionalidad es gestionar los servicios en el sistema, así como registrar la actividad que generen dichos servicios.

Lo primero, por tanto, será arrancar el servicio IceBox, al que tendremos que indicar que queremos hacer uso del servicio **IceStorm**.

Fichero config.icebox:

```
//Indicamos el host en el que correrá icebox. En nuestro caso, en
//localhost y el puerto 10030.
IceBox.ServiceManager.Endpoints=tcp -p 10030

//A continuación le indicamos que queremos crear un servicio IceStorm,
//y además le indicamos cual será el fichero de configuración de dicho
//servicio. De este modo IceBox realiza las operaciones pertinentes para
//crear y gestionar el servicio IceStorm, indicándole a éste cual será su
//configuración.

IceBox.Service.IceStorm=IceStormService,33:createIceStorm –
Ice.Config=config.service
```

Fichero config.service:

```
//Indicamos primeramente como haremos referencia al servicio IceStorm,
//sin complicarnos demasiado asignamos como nombre “IceStorm”. Este
//nombre es importante a la hora de referenciar a otras propiedades.

IceStorm.InstanceName=IceStorm

//En esta línea estamos indicando que el TopicManager del servicio, al
//que accederán tanto publishers como subscribers, se encuentra en la
//máquina definida y en el puerto definido. En este punto cabe destacar
//que este host será el de por defecto, pero es rápidamente configurable,
//puesto que como veremos el código está preparado para que en la
//ejecución indiquemos por parámetro cual es la IP del host donde se
//encuentra el TopicManager, es decir donde está corriendo el servicio
//IceStorm. No obstante, esta línea sirve de refuerzo y de referencia.

IceStorm.TopicManager.Endpoints=tcp -h 163.117.141.38 -p
10010
```

```
//Esta línea tiene suma importancia pues es donde indicamos en qué
//puerto se escuchará al tópico de los publishers. Es decir, un publisher
//enviará su información al servicio IceStorm al puerto que figure
//configurado en esta línea.

IceStorm.Publish.Endpoints=tcp -p 10020:udp -p 10020


//En esta línea estamos definiendo un nivel de detalle 2 (el más
//detallado) para toda la actividad que genere el TopicManager, de esta
//forma esta información nos sirve para entender posibles problemas de
//comunicación que puedan producirse.

IceStorm.Trace.TopicManager=2


//Esta línea es equivalente a la anterior, pero con un nivel de detalle más
//bajo y para detallar la actividad del subscriber. Se decidió detallar la del
//subscriber y no la del publisher, debido a que normalmente es el
//subscriber el primer elemento de la comunicación en interactuar con el
//servicio IceStorm (en la aplicación de videovigilancia concreta), y de
//este modo accediendo a la información que generase esta propiedad
//sería más rápidamente visible cualquier error que existiera al establecer
//la configuración del TopicManager del servicio IceStorm.

IceStorm.Trace.Subscriber=1

//Finalmente encontramos esta línea en la cual se indica el directorio en
//el que va a guardarse la actividad generada en el servicio IceStorm, es
//decir referencias a los topicos existentes en el TopicManager, accesos,
//etc, etc, información que puede ser útil para rastrear actividades en el
//sistema. Se indica como directorio “db”, de modo que es necesario que
//exista dicho directorio antes de arrancar el servicio.

Freeze.DbEnv.IceStorm.DbHome=db
```

Los ficheros comentados anteriormente incluyen algunas líneas de configuración más, que no han sido comentadas debido a que no son relevantes.

Anexo 4: Aplicación de videovigilancia

El presente anexo tiene como objetivo mostrar el aspecto que presentan las interfaces graficas de la aplicación, del lado de cliente. El objetivo es ver de un modo grafico la funcionalidad que se ha implementado.

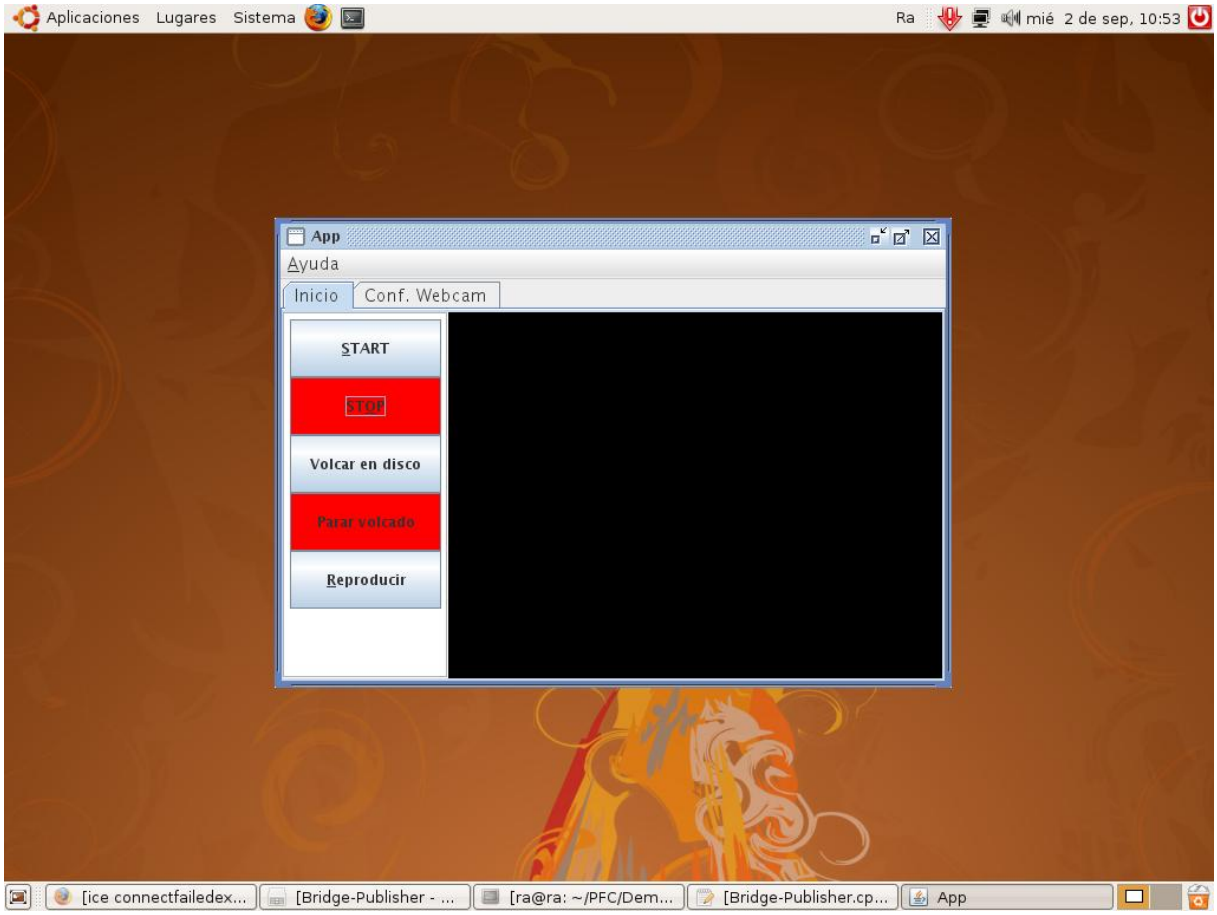


Figura 34: Aplicación de videovigilancia parada.

En esta captura se observa el aspecto de la aplicación parada, donde puede apreciarse:

- **Dos pestañas**: La de inicio es la que se muestra, la pestaña llamada Conf. Webcam contiene los controles de la webcam, para configurar ésta.

- **Botón START**: Arranca la captura, envío y muestra de las

imágenes capturadas. En la maquina local se muestran en el recuadro negro que podemos observar.

- **Botón STOP**: Para la captura, muestra y envió de datos.
- **Botón Volcar en Disco**: Comienza a volcar la información a disco.
- **Botón Parar volcado**: Deja de volcarse información y se muestra un cuadro al usuario preguntando por el lugar y nombre del archivo que quiere guardarse.
- **Botón Reproducir**: Ofrece un cuadro de dialogo donde el usuario puede seleccionar el archivo a reproducir.

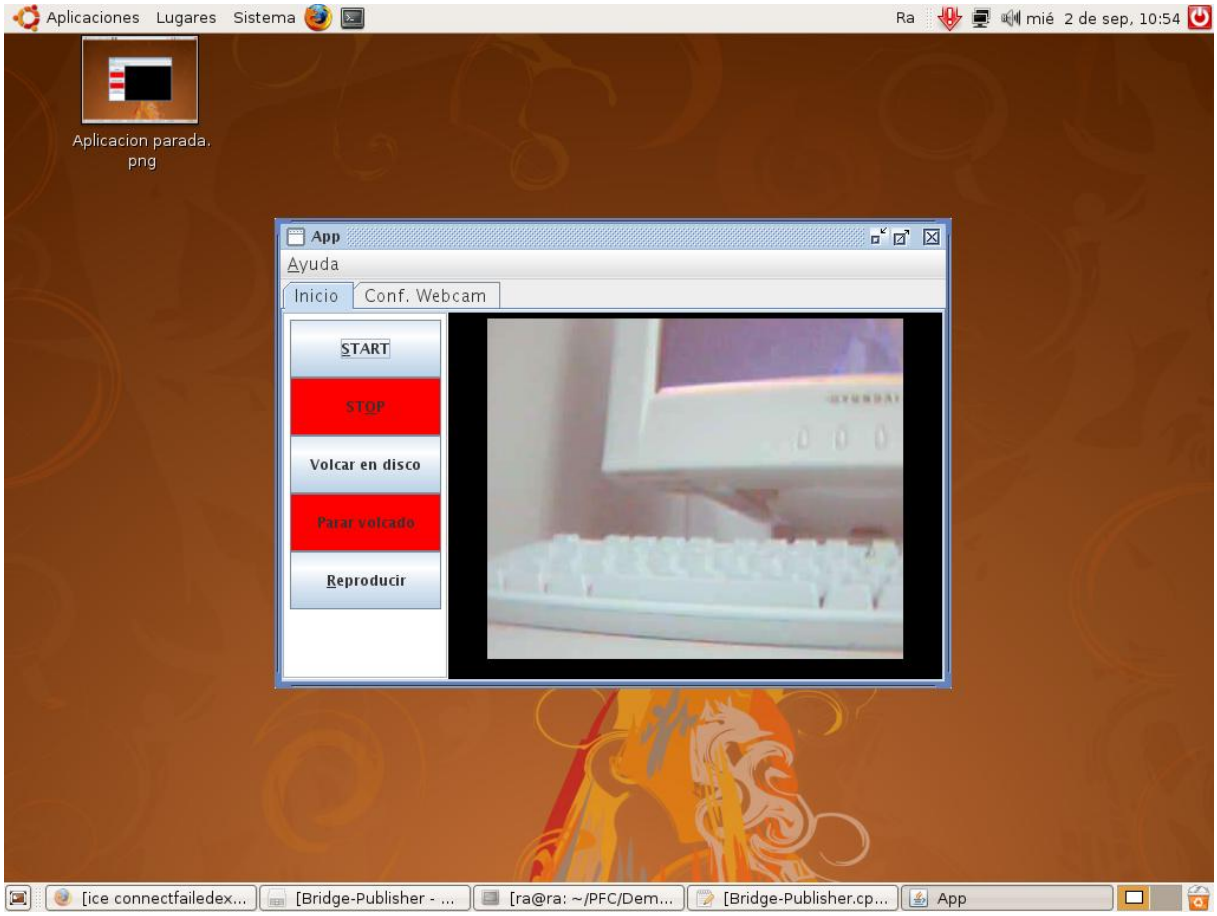


Figura 35: Aplicación de videovigilancia en funcionamiento.

La captura anterior muestra la aplicación funcionando

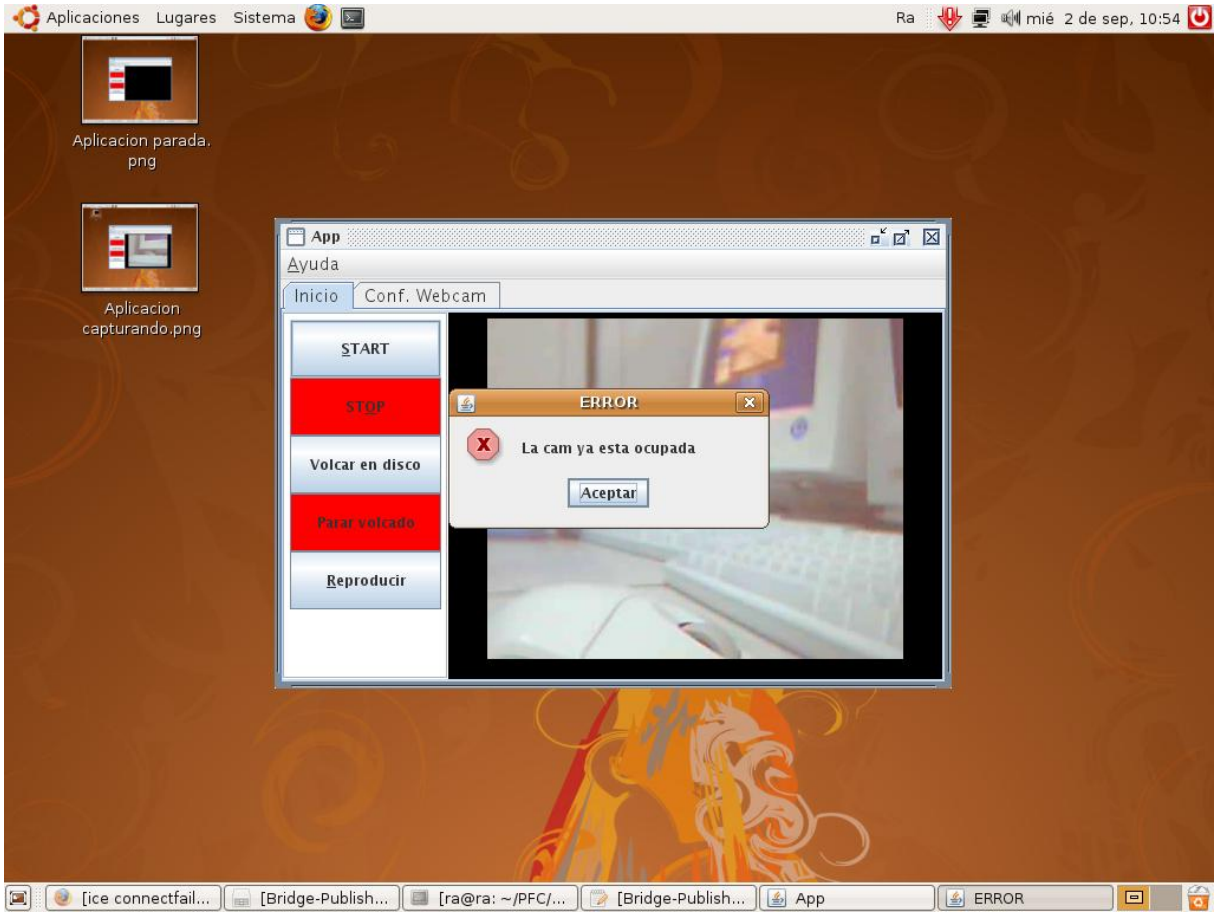


Figura 36: Aplicación de videovigilancia. Error.

En la anterior, se ha pulsado el botón START una vez la webcam ya estaba funcionando, se muestra un cuadro de diálogo donde se muestra el error. El control de errores está cuidado en toda la interfaz gráfica. De este modo, no podemos parar la cámara cuando ya está parada, ni arrancar la aplicación cuando ya está funcionando, ni semejantes acciones con el volcado a disco.

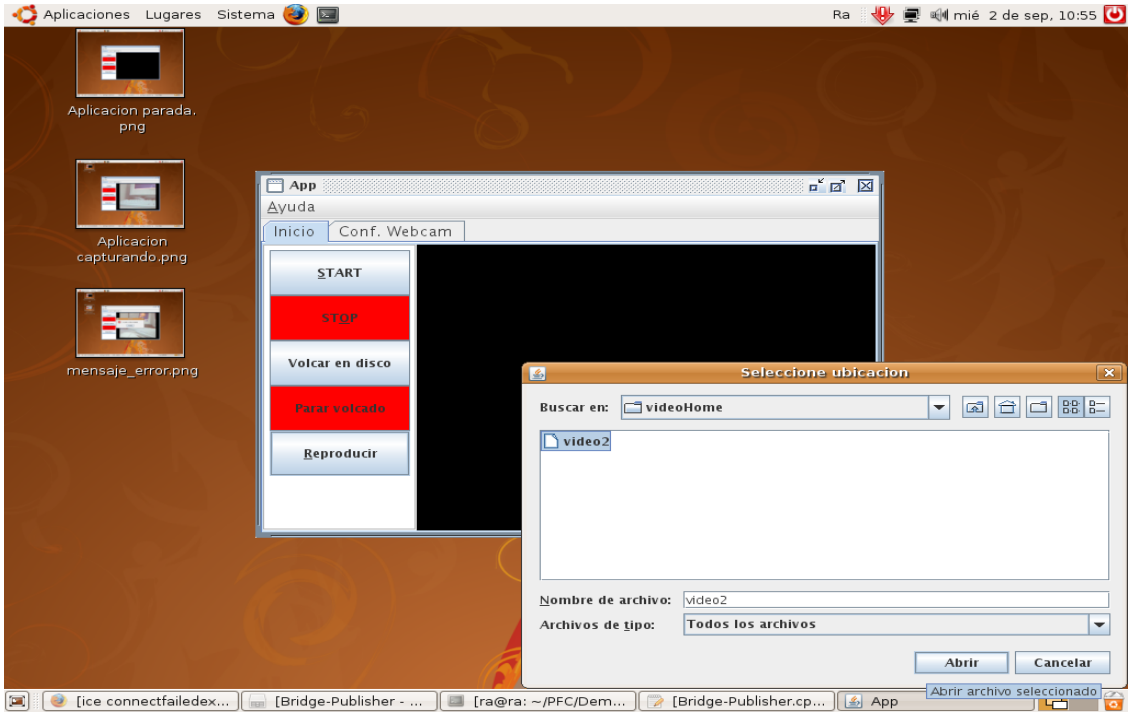
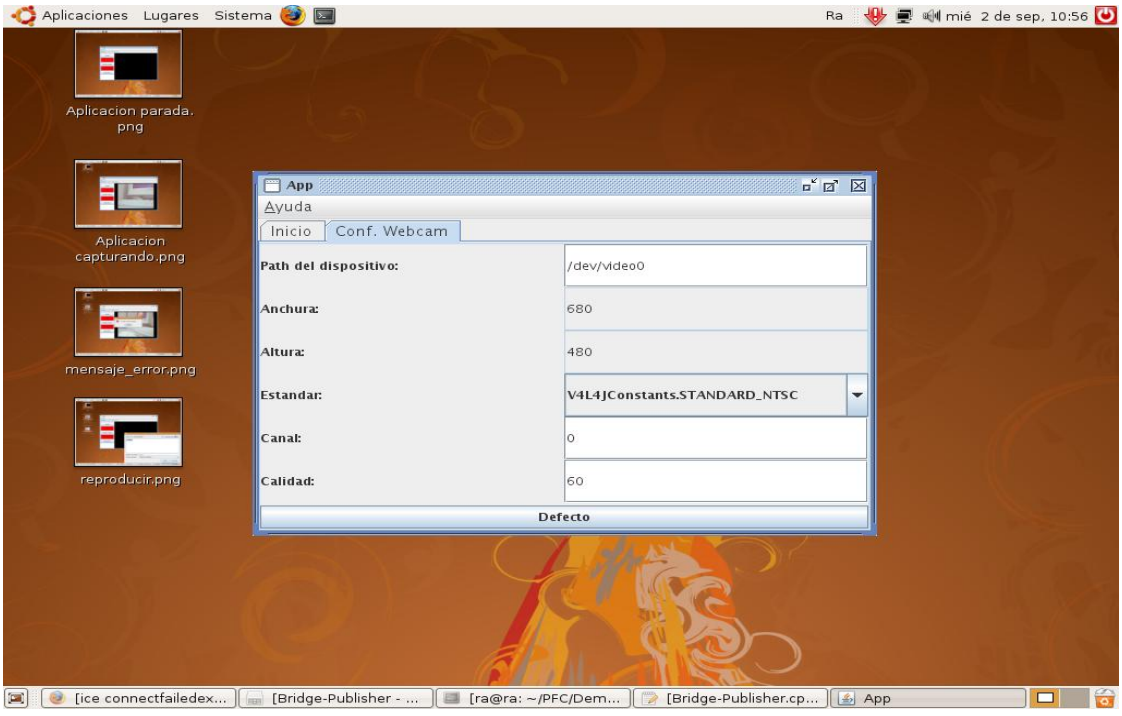


Figura 37: Aplicación de videovigilancia. Reproducir.

En la anterior se ha pulsado el botón Reproducir, y se muestra un cuadro donde se da la opción de elegir un fichero previamente volcado a disco, en este caso existe el fichero “video2”.



Anexo 5: Código de aplicaciones C++/Java evaluación

En el presente apéndice se muestra el código de los *publishers* diseñados para evaluar **IceStorm** (capitulo 4). Se mostrará como ejemplo el implementado en **C++**, así como el *subscriber* correspondiente. Ambos comentados y explicados. Observando el código en **Java**, se ve rápidamente que partes corresponden entre ambas implementaciones, y es por esto que se ha decidido mostrar solo una implementación, para evitar redundancia. También se muestra la aplicación cliente y la servidora, desarrolladas para realizar las pruebas de RTT y RTT Localhost. Finalmente se muestran las capturas que muestran la anomalía que se produce en la prueba RTT con interferencia de memoria, y que se comenta en la página 158.

Publisher C++:

```
//Librerías correspondientes a Ice e IceStorm

#include <Ice/Ice.h>
#include <IceStorm/IceStorm.h>

//El header del código implementado en Slice.

#include <Send.h>

//Librería de io estándar de C++

#include <iostream>

//Librería de hilos POSIX de C.

#include <pthread.h>

//Librerias relativas al control de tiempo en C/C++.
#include <sys/time.h>
```

```
#include <time.h>

using namespace std;
using namespace eval;

//Esqueleto de la función encargada de realizar la interferencia en memoria.

void *MemInterference(void *t);

//Esqueleto de la función encargada de realizar la interferencia en cpu.

void *CPUInterference(void *t);

//Función encargada de devolver la diferencia de tiempos entre dos estructuras que
//guardan dicho tiempo.

double timeval_diff(struct timeval *a, struct timeval *b){
          return (double)(a->tv_sec + (double)a->tv_usec/1000000) -
(double)(b->tv_sec + (double)b->tv_usec/1000000);
}

//Método main de la aplicación.

int main(int argc, char* argv[]){

          //Parte relativa a la inicialización de IceStorm. Comentado
anteriormente.

          Ice::CommunicatorPtr ic;
          ic = Ice::initialize();

          //Se presupone un servicio IceStorm corriendo en un host dado.

          Ice::ObjectPrx obj = ic-
>stringToProxy("IceStorm/TopicManager:tcp -h XXX.XXX.XXX.XXX -p
9999");
          IceStorm::TopicManagerPrx topicManager =
IceStorm::TopicManagerPrx::checkedCast(obj);
          IceStorm::TopicPrx topic;
```

```
            //Tras inicializar la comunicación se accede al tópico, que se ha llamado
            //Data.

            try{
            topic = topicManager->retrieve("Data");
            }
            catch(const IceStorm::NoSuchTopic&) {
            topic = topicManager->create("Data");
            }
            Ice::ObjectPrx pub = topic->getPublisher()->Ice_oneway();

            //Se hace el casting al objeto send, la interfaz con las operaciones
            //disponibles.

            SendPrx send = SendPrx::uncheckedCast(pub);

            //Tras realizar la inicialización de la comunicación, se comprueban los
            //parámetros que se reciben por línea de comandos.

            if(argc != 3){
                        cout << "Uso: " << endl;
                        cout << "Arg1 -> dimension de los datos a
enviar." << endl;
                        cout << "Arg2 -> mem-> interferencia de memoria,
cpu-> interferencia de cpu. " << endl;
                        exit(1);
            }

            //Si todo está bien se crea toma la longitud del payload introducida.

            int dim = atoi(argv[1]);
            int t = 0;
            int id = 0;

            //Y se crea un vector con tantas posiciones con dimensión el payload.

            vector<Ice::Byte> data (dim,5);

            //Se toma el comando introducido por línea de comandos.
```

```
		string command (argv[2]);
		string mem ("mem");
		string cpu ("cpu");
		string x ("x");
		string free ("free");

		//Y se comprueba cual ha sido…

		if(command.compare(mem) == 0){

			//Si es interferencia de memoria, se inicializan los parámetros
			//del hilo y se arranca este, pasándole por parámetro la
			//función que debe ejecutar en paralelo.

			pthread_t memInt;
			id = pthread_create(&memInt, NULL,
MemInterference, (void *)t);

		}
		else if(command.compare(cpu) == 0){

			//Si es interferencia de cpu, se inicializan los parámetros del
			//hilo y se arranca, pasándole por parámetro la función que
			//debe ejecutar en paralelo.

			pthread_t cpuInt;
			id = pthread_create(&cpuInt, NULL,
CPUInterference, (void *)t);
		}

		//Caso especial de prueba no tratado.

		else if(command.compare(x)){
			pthread_t memInt;
			id = pthread_create(&memInt, NULL,
MemInterference, (void *)t);
			pthread_t cpuInt;
			id = pthread_create(&cpuInt, NULL,
CPUInterference, (void *)t);
		}
```

```
//En el caso de introducir free (libre de interferencia), simplemente no se hace nada.

else if(command.compare(free)){
}
else{
        cout << "Arg2 -> mem o cpu" << endl;
}

//Las siguientes instrucciones se añaden para ofrecer al usuario un
//modo de comenzar la prueba cuando introduzca un carácter y pulse
//retorno.

char a;
cin >> a;

//Se crean las estructuras que recogerán el tiempo inicial y final.

timeval t_ini, t_end;
double dif;
int count = 0;
cout << "Iteracion, Delay" << endl;
int j = 0;

//Para la prueba en la que se mide el tiempo total de envió de 10000 se
//utilice un for, que realice la operación 20 veces, para conseguir más
//tarde una media.
//TOTAL TIME ***

for(j = 0; j<20; j++){

//Se toma el tiempo al inicio de la prueba.

gettimeofday(&t_ini, NULL);

//Se realice la prueba.

while(count < 10000) {
        send->report(data);
```

```
                count++;
        }

        //Se toma el tiempo al final de la prueba.

        gettimeofday(&t_end, NULL);

        //Se halla la diferencia, es decir el tiempo total de la prueba.

        dif = timeval_diff(&t_end, &t_ini);

        //Finalmente se introduce tal diferencia por línea de comandos.

        cout << j << "," << dif << endl;
        count = 0;

        //TOTAL TIME ***

        }
}

//La funcion encargada de generar la interferencia de cpu.

void *CPUInterference(void *t){

        //Se crea un fichero

        FILE *nullFile;

        //Se entra en un bucle infinito, que consiga mantener la cpu ocupada
        //con las sentencias que encontramos dentro.

        while(1){

                //Se abre el fichero en /dev/null, que descartara lo que le
                //llegue.

                nullFile = fopen("/dev/null", "w+");

                //Se imprime en este la cadena de texto siguiente.
```

```
                    fprintf(nullFile,
"fñlasjdfñlasjdñflasjdñflasjdñfljajaalñskdjfaklñsjfdlñasjfdlñasjdfff
fffdf");

                    //Se cierra el fichero, para consumer aun mas recursos.

                    fclose(nullFile);
          }
}


//La funcion encargada de realizar la interferencia de memoria.

void *MemInterference(void *t){

          //Se calculan los bytes que hay en 1mb, y se multiplicand estos por 90.
          //Esta cantidad ha sido ajustada empíricamente, siendo la cantidad que
          //consigue un mayor de consumo de memoria para el caso de la
          //aplicación en C++.

          int dim = 90*1048576;                    //x * 1MB

          //Se inicializan varios vectores con dicha cantidad.

          vector<char> a (dim, 'a');
          vector<char> b (dim, 'b');
          vector<char> c (dim, 'c');
          vector<char> d (dim, 'd');
          vector<char> e (dim, 'e');
          vector<char> f (dim, 'f');
}
```

En el ejemplo anterior, se ha mostrado la funcionalidad en el caso de realizar la prueba de tiempo total de envío de 10000 muestras, (código entre los comentarios TOTAL TIME ***). Para el caso del tiempo de Marshall, el bucle quedaría de esta forma.

```
          //Se toman 10000 muestras igualmente
```

```
while(count < 10000) {

        //Se toma el tiempo al inicio de la operación.

        gettimeofday(&t_ini, NULL);

        //Se envian los datos.

        send->report(data);
        count++;

        //Se toma el tiempo al final de la operación

        gettimeofday(&t_end, NULL);

        //Se calcula la diferencia

        dif = timeval_diff(&t_end, &t_ini);

        //Se imprimen los resultados.

        cout << count << "," << dif << endl;
    }
```

Subscriber C++:

```
//Del mismo modo que en la parte del publisher, se incluyen las librerias
//necesarias.

#include <Ice/Ice.h>
#include <IceStorm/IceStorm.h>
#include <Send.h>

using namespace std;
using namespace eval;

//Variable que llevara la cuenta de los datos que han llegado.
```

```
        int i = 0;

        //Se crea la clase SendI, que hereda publicamente de Send

        class SendI : virtual public Send {
                public:
                //Prototipo virtual de la funcionalidad que implementara el
                //servant de la comunicación.

                virtual void report(const Data& data, const
Ice::Current&);
        };

        //Implementación del método report, que en este caso no hace más que
        //presentar el número de dato que ha llegado.

        void SendI::report(const eval::Data& data, const
Ice::Current&){
                cout << "Datos: " << endl;
                cout << i << endl;
                i++;
        }

        //Método main de la aplicación

        int main(int argc, char* argv[]){

                //Se inicializan los parámetros de la comunicación.

                Ice::CommunicatorPtr ic;
                ic = Ice::initialize();

                //Para que la aplicación de prueba funcione, el topicManager
                //de la comunicación, o dicho de otro modo, el host donde
                //corre el servicio IceStorm debe conocerse tanto por
                //Publisher como por subscriber.

                Ice::ObjectPrx obj = ic-
>stringToProxy("IceStorm/TopicManager:tcp -h XXX.XXX.XXX.XXX -p
```

```
9999");
                    IceStorm::TopicManagerPrx topicManager =
IceStorm::TopicManagerPrx::checkedCast(obj);
                    Ice::ObjectAdapterPtr adapter = ic-
>createObjectAdapterWithEndpoints("DataAdapter", "default");
                    SendPtr send = new SendI;
                    Ice::ObjectPrx proxy = adapter-
>addWithUUID(send)->Ice_oneway();
                    IceStorm::TopicPrx topic;

                    //Tras inicializar los parámetros principales se accede al topic
                    //Data.

                    try {
                              topic = topicManager->retrieve("Data");
                              IceStorm::QoS qos;
                              topic->subscribeAndGetPublisher(qos,
proxy);
                    }
                    catch (const IceStorm::NoSuchTopic&) {
                              // Error! No topic found!
                    }

                    //Finalmente se active el adaptador, para que el servant que
                    //implementa en esta ocasión la funcionalidad pueda recibir
                    //peticiones.

                    adapter->activate();

                    //Simplemente se espera hasta que fuercen el final de la
                    //aplicación.

                    ic->waitForShutdown();
                    topic->unsubscribe(proxy);
          }
```

Aplicación Cliente para RTT:

En este caso se muestran partes parciales del código puesto que algunas ya han sido detalladas anteriormente:

```
//Primeramente se definen las variables globales que van a utilizarse. Las estructuras timeval, serán en las que se guarde el tiempo. La variable counter se encargara de llevar la cuenta de los 10000 envios. Y el proxy send el utilizado para enviar los datos al otro nodo.

timeval t_ini, t_end;
int counter;
SendPrx send;

//Se definen los prototipos de las funciones que van a utilizarse, y la estructura timeval.

void exec(vector<Ice::Byte> data);

double timeval_diff(struct timeval *a, struct timeval *b){
	return (double)(a->tv_sec + (double)a->tv_usec/1000000) -
(double)(b->tv_sec + (double)b->tv_usec/1000000);
}

class SendI : virtual public Send{
	public:
		virtual void report(const Data& data, const
Ice::Current&);
};

//A continuación la implementación del método report, que recoge las respuestas del otro nodo y captura el tiempo de llegada.

void SendI::report(const eval::Data& data, const Ice::Current&){
	double dif = 0;

		//Se toma el tiempo de llegada.
```

```
            gettimeofday(&t_end, NULL);
            //Se calcula la diferencia entre el momento que enviamos y el que llego

            dif = timeval_diff(&t_end, &t_ini);

            //Se imprime dicha diferencia

            cout << counter << "," << dif << endl;

            //Si el contador es menor que 10000, entonces volvemos a enviar los
datos.

            if(counter < 10000) exec(data);
}

//Mas prototipos, esta vez de los hilos. Interferencia de memoria, de cpu, y
InitSubscriber es el hilo que recoge la funcionalidad del subscriber en el lado cliente.

void *MemInterference(void *t);

void *CPUInterference(void *t);

void *InitSubscriber(void *t);

                    […]

//Se inicializa el hilo que crea el subscriber de parte del cliente.

int id_subscriber = 0;
int t = 0;
pthread_t subscriberT;
id_subscriber = pthread_create(&subscriberT, NULL, InitSubscriber,
(void *)t);
                    […]

//En el programa principal se controla cuando quiere que comience la ejecución con
la introducción de un character por consola.

char a;
cin >> a;
```

```
cout << "Iteracion, Delay" << endl;
counter = 0;
```

//Y tras esto se realiza el primer envío, tras lo cual se coloca una espera active para que no termine la ejecución del programa principal, lo que supondría la parada de la aplicación completa.

```
exec(data);
while(1);
```

[…]

//El método exec, recibe unos datos, que envía al otro nodo tras captar el instante de tiempo actual. Tras esto incrementa el contador, para controlar el número de muestras enviadas.

```
void exec(vector<Ice::Byte> data){
          gettimeofday(&t_ini, NULL);
          send->report(data);
          counter++;
}
```

//El hilo InitSubscriber implementa el subscriber del lado del cliente, como puede verse en el código. El subscriber se bloquea en la operación waitForShutdown(), por lo que es necesario incluir dicha funcionalidad en un hilo.

```
void *InitSubscriber(void *t){
          Ice::CommunicatorPtr icS;
          icS = Ice::initialize();
          Ice::ObjectPrx objS = icS-
>stringToProxy("IceStorm/TopicManager:tcp -h 163.117.141.44 -p
22222");
          IceStorm::TopicManagerPrx topicManagerS =
          IceStorm::TopicManagerPrx::checkedCast(objS);
          Ice::ObjectAdapterPtr adapter = icS-
>createObjectAdapterWithEndpoints("DataRETURNAdapter", "default");
          SendPtr receive = new SendI;
          Ice::ObjectPrx proxyS = adapter->addWithUUID(receive)-
>ice_oneway();
          IceStorm::TopicPrx topicS;
```

```
        try{
                topicS = topicManagerS->retrieve("DataRETURN");
                IceStorm::QoS qos;
                topicS->subscribeAndGetPublisher(qos, proxyS);
        }
        catch(const IceStorm::NoSuchTopic&){
        }
        adapter->activate();
        icS->waitForShutdown();
        topicS->unsubscribe(proxyS);
}
```

Aplicación Servidor para RTT:

De la misma forma que para el cliente, las partes comentadas anteriormente se omiten.

```
//Se configuran las variables globales que van a utilizarse en la aplicación. La variable i cuenta los datos que van llegando. El proxy receive es el correspondiente al publisher del lado de servidor, que se utiliza para reenviar los datos a la aplicación cliente.

int i = 0;
SendPrx receive;

//Prototipo de la función initPublisher() que dará como resultado el proxy declarado más arriba.

void initPublisher();

class SendI : virtual public Send {
        public:
                virtual void report(const Data& data, const
Ice::Current&);
};

//La implementación del método report como se aprecia, consiste en reenviar los datos que han llegado al primer nodo, tras lo cual se imprime  una cadena de texto
```

informativa.

```
void SendI::report(const eval::Data& data, const Ice::Current&){
          receive->report(data);
          cout << "Datos: " << endl;
          cout << i << endl;
          i++;
}
          […]
```

//El método initPublisher crea e inicializa una comunicación hacia el primer nodo, de modo que consigue un proxy que es el utilizado para reenviar los datos. Este sentido de la comunicación utiliza un topic llamado “DataRETURN”.

```
void initPublisher(){
          Ice::CommunicatorPtr icP;
          icP = Ice::initialize();
          Ice::ObjectPrx objP = icP-
>stringToProxy("IceStorm/TopicManager:tcp -h 163.117.141.44 -p
22222");
          IceStorm::TopicManagerPrx topicManagerP =
IceStorm::TopicManagerPrx::checkedCast(objP);
          IceStorm::TopicPrx topicP;
          try{
                    topicP = topicManagerP->retrieve("DataRETURN");
          }
          catch(const IceStorm::NoSuchTopic&){
                    topicP = topicManagerP->create("DataRETURN");
          }
          Ice::ObjectPrx pubP = topicP->getPublisher()->ice_oneway();
          receive = SendPrx::uncheckedCast(pubP);
}
```

Anomalías de la aplicación de prueba RTT (referenciada en pag 158):

Durante la realización de las pruebas de RTT con interferencia de memoria se observa como hacia la mitad de muestras enviadas (unas 5000) la interferencia de memoria que se emula en el sistema desaparece, y se vuelven a

valores de memoria normales, tras lo cual, obviamente el rendimiento de la aplicación mejora notablemente. La siguiente captura de un monitor de recursos del sistema operativo muestra el efecto:

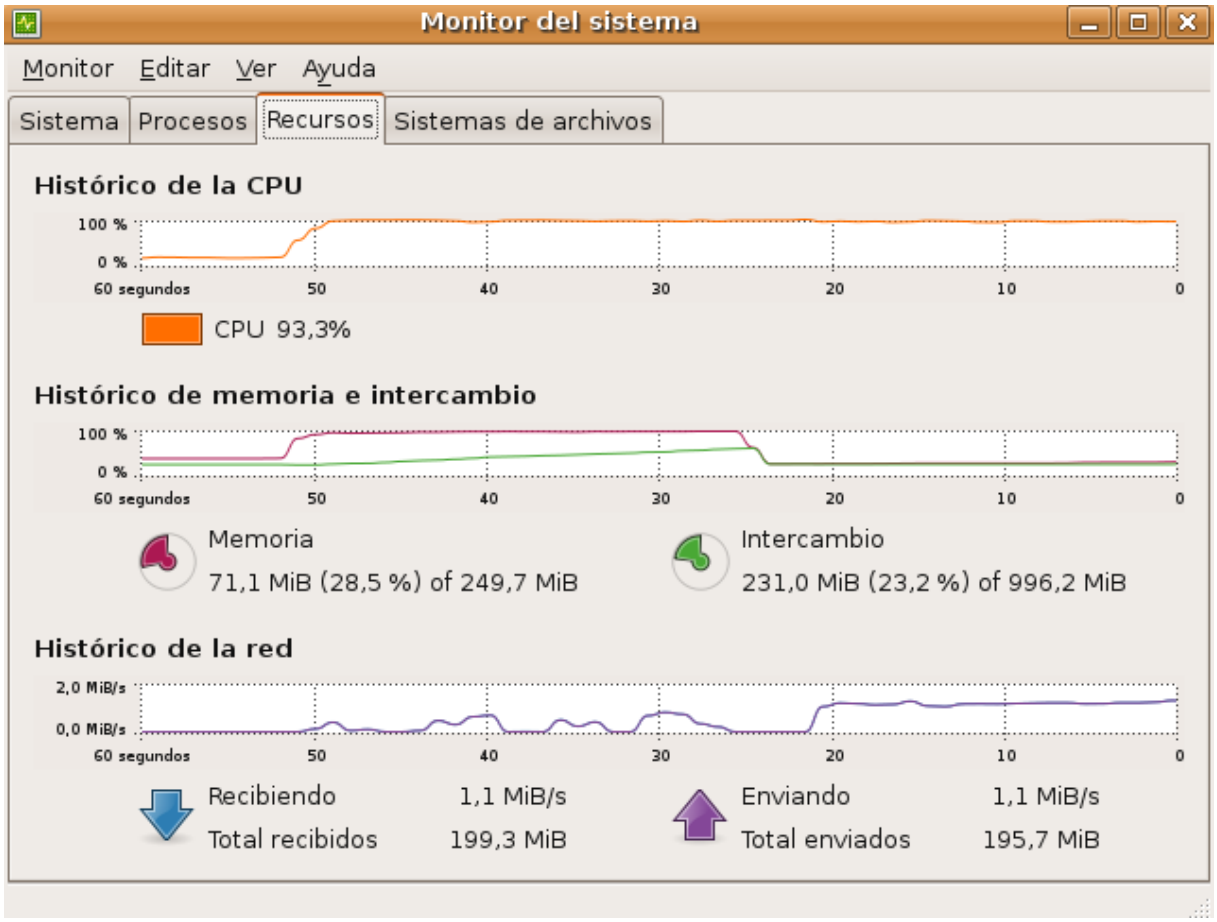


Figura 38: Anomalía de memoria en prueba RTT distribuida.

Bibliografía y Enlaces

[1] Sitio oficial de OMG

www.omg.org

[2] Documentos oficiales de las especificaciones de DDS

http://www.omg.org/technology/documents/**DDS**_spec_catalog.htm

[3] OMG Available Specification formal/07-01-01 “Data Distribution Service for Real Time Systems Version 1.2”. OMG.

[4] “The Real Time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification Version 2.1” OMG Document Number: formal:2009-01-05.

[5] Sitio oficial de la tecnología DDS del OMG.

http://portals.omg.org/**DDS**/OMG_Data_Distribution_Portal

[6] Sitio oficial de la compañía ZeroC:

http://www.zeroc.com/

[7] “*Distributed Programming with Ice*” Miki Henning, Mark Spruiell. Revision 3.3.1, Julio 2009.

[8] Sitio oficial de RTI DDS:

http://www.rti.com/products/**DDS**/index.html

[9] Sitio oficial de ICE-E

http://www.zeroc.com/icee/index.html

[10] Página con información variada de sistemas distribuidos embarcados

http://www.embedded.com/

[11] Sitio de la libreria v4l4j

http://code.google.com/p/v4l4j/

[12] API de Java 1.6

http://java.sun.com/javase/6/docs/api/

[13] Referencias a C++:

http://www.cplusplus.com/doc/tutorial/

http://www.conclase.net/c/

http://www.research.att.com/~bs/C++.html

[14] Recursos sobre hilos POSIX:

http://en.wikipedia.org/wiki/POSIX_Threads

http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html

https://computing.llnl.gov/tutorials/pthreads/

[15] Pagina oficial wikipedia en ingles.

http://en.wikipedia.org/wiki/Main_Page

[16] Apuntes de la asignatura: Arquitecturas Distribuidas.

[17] Recursos sobre sistemas operativos en tiempo real:

QNX: http://www.qnx.com/

vxWorks: http://www.windriver.com/

INTEGRITY: http://www.ghs.com/

[18] Sobre DO-178B

http://en.wikipedia.org/wiki/DO-178B

http://www.savive.com/external/rtca/do178/index.html

[19] "*Real-Time Systems and Programming Languages*" Alan Burns and Andy Wellings.

http://books.google.es/books?id=0_LjXnAN6GEC&dq=real+time+systems+and+programming+languages&printsec=frontcover&source=bn&hl=es&ei=FXSlSpTuC8-fjAfmmdG7Dg&sa=X&oi=book_result&ct=result&resnum=4#v=onepage&q=&f=false

http://www.cs.york.ac.uk/rts/books/RTSBookThirdEdition.html